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.
INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. INTEL PRODUCTS ARE NOT INTENDED FOR USE IN MEDICAL, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS.

Intel may make changes to specifications and product descriptions at any time, without notice.

Developers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Improper use of reserved or undefined features or instructions may cause unpredictable behavior or failure in developer's software code when running on an Intel processor. Intel reserves these features or instructions for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from their unauthorized use.

The Intel® IA-32 architecture processors (e.g., Pentium® 4 and Pentium III processors) may contain design defects or errors known as errata. Current characterized errata are available on request.

Hyper-Threading Technology requires a computer system with an Intel® Pentium® 4 processor supporting Hyper-Threading Technology and an HT Technology enabled chipset, BIOS and operating system. Performance will vary depending on the specific hardware and software you use. See http://www.intel.com/info/hyperthreading/ for more information including details on which processors support HT Technology.

Intel, Intel386, Intel486, Pentium, Intel Xeon, Intel NetBurst, Intel SpeedStep, OverDrive, MMX, Celeron, and Itanium are trademarks or registered trademarks of Intel Corporation and its subsidiaries in the United States and other countries.

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

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

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation
P.O. Box 5937
Denver, CO 80217-9808

or call 1-800-548-4725
or visit Intel’s website at http://www.intel.com

Copyright © 1997 - 2004 Intel Corporation
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**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /3</td>
<td>NEG r/m8</td>
<td>Two’s complement negate r/m8.</td>
</tr>
<tr>
<td>F7 /3</td>
<td>NEG r/m16</td>
<td>Two’s complement negate r/m16.</td>
</tr>
<tr>
<td>F7 /3</td>
<td>NEG r/m32</td>
<td>Two’s complement negate r/m32.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>NOP</td>
<td>No operation.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /2</td>
<td>NOT r/m8</td>
<td>Reverse each bit of r/m8.</td>
</tr>
<tr>
<td>F7 /2</td>
<td>NOT r/m16</td>
<td>Reverse each bit of r/m16.</td>
</tr>
<tr>
<td>F7 /2</td>
<td>NOT r/m32</td>
<td>Reverse each bit of r/m32.</td>
</tr>
</tbody>
</table>

**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

### 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.
**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 while the current privilege level is 3.

#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

**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**

\[
\text{DEST}[127-0] \leftarrow \text{DEST}[127-0] \text{ BitwiseOR SRC}[127-0];
\]

**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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 56 /r</td>
<td>ORPS xmm1, xmm2/m128</td>
<td>Bitwise OR of xmm2/m128 and xmm1.</td>
</tr>
</tbody>
</table>

**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

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 I, 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;

Opcode Instruction Description
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E6 ib</td>
<td>OUT imm8, AL</td>
<td>Output byte in AL to I/O port address imm8.</td>
</tr>
<tr>
<td>E7 ib</td>
<td>OUT imm8, AX</td>
<td>Output word in AX to I/O port address imm8.</td>
</tr>
<tr>
<td>E7 ib</td>
<td>OUT imm8, EAX</td>
<td>Output doubleword in EAX to I/O port address imm8.</td>
</tr>
<tr>
<td>EE</td>
<td>OUT DX, AL</td>
<td>Output byte in AL to I/O port address in DX.</td>
</tr>
<tr>
<td>EE</td>
<td>OUT DX, AX</td>
<td>Output word in AX to I/O port address in DX.</td>
</tr>
<tr>
<td>EE</td>
<td>OUT DX, EAX</td>
<td>Output doubleword in EAX to I/O port address in DX.</td>
</tr>
</tbody>
</table>
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

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

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

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

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**

\[
\begin{align*}
\text{DEST}[7..0] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ DEST}[15..0]; \\
\text{DEST}[15..8] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ DEST}[31..16]; \\
\text{DEST}[23..16] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ DEST}[47..32]; \\
\text{DEST}[31..24] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ DEST}[63..48]; \\
\text{DEST}[39..32] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ SRC}[15..0]; \\
\text{DEST}[47..40] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ SRC}[31..16]; \\
\text{DEST}[55..48] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ SRC}[47..32]; \\
\text{DEST}[63..56] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ SRC}[63..48];
\end{align*}
\]

**PACKSSDW instruction with 64-bit operands**

\[
\begin{align*}
\text{DEST}[15..0] &\leftarrow \text{SaturateSignedDoublewordToSignedWord} \text{ DEST}[31..0]; \\
\text{DEST}[31..16] &\leftarrow \text{SaturateSignedDoublewordToSignedWord} \text{ DEST}[63..32]; \\
\text{DEST}[47..32] &\leftarrow \text{SaturateSignedDoublewordToSignedWord} \text{ SRC}[31..0]; \\
\text{DEST}[63..48] &\leftarrow \text{SaturateSignedDoublewordToSignedWord} \text{ SRC}[63..32];
\end{align*}
\]

**PACKSSWB instruction with 128-bit operands**

\[
\begin{align*}
\text{DEST}[7..0] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ (DEST}[15..0]); \\
\text{DEST}[15..8] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ (DEST}[31-16]); \\
\text{DEST}[23..16] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ (DEST}[47-32]); \\
\text{DEST}[31..24] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ (DEST}[63-48]); \\
\text{DEST}[39..32] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ (DEST}[79-64]); \\
\text{DEST}[47..40] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ (DEST}[95-80]); \\
\text{DEST}[55..48] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ (DEST}[111-96)]; \\
\text{DEST}[63..56] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ (DEST}[127-112]); \\
\text{DEST}[71..64] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ (SRC}[15..0]); \\
\text{DEST}[79..72] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ (SRC}[31-16]); \\
\text{DEST}[87..80] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ (SRC}[47-32]); \\
\text{DEST}[95..88] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ (SRC}[63-48]); \\
\text{DEST}[103..96] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ (SRC}[79-64]); \\
\text{DEST}[111..104] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ (SRC}[95-80]); \\
\text{DEST}[119..112] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ (SRC}[111-96]); \\
\text{DEST}[127..120] &\leftarrow \text{SaturateSignedWordToSignedByte} \text{ (SRC}[127-121]);
\end{align*}
\]

**PACKSSDW instruction with 128-bit operands**

\[
\begin{align*}
\text{DEST}[15..0] &\leftarrow \text{SaturateSignedDwordToSignedWord} \text{ (DEST}[31-0]);
\end{align*}
\]
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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 67</td>
<td>PACKUSWB mm, mm/m64</td>
<td>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.</td>
</tr>
<tr>
<td>66 0F 67</td>
<td>PACKUSWB xmm1, xmm2/m128</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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:

```assembly
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:

```assembly
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]);
```
INSTRUCTION SET REFERENCE, N-Z

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. Execu-
tion 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.
INSTRUCTION SET REFERENCE, N-Z

PADDB/PADDW/PADDD—Add Packed Integers

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F FC /r</td>
<td>PADDB mm, mm/m64</td>
<td>Add packed byte integers from mm/m64 and mm.</td>
</tr>
<tr>
<td>66 0F FC /r</td>
<td>PADDB xmm1,xmm2/m128</td>
<td>Add packed byte integers from xmm2/m128 and xmm1.</td>
</tr>
<tr>
<td>0F FD /r</td>
<td>PADDW mm, mm/m64</td>
<td>Add packed word integers from mm/m64 and mm.</td>
</tr>
<tr>
<td>66 0F FD /r</td>
<td>PADDW xmm1, xmm2/m128</td>
<td>Add packed word integers from xmm2/m128 and xmm1.</td>
</tr>
<tr>
<td>0F FE /r</td>
<td>PADDD mm, mm/m64</td>
<td>Add packed doubleword integers from mm/m64 and mm.</td>
</tr>
<tr>
<td>66 0F FE /r</td>
<td>PADDD xmm1, xmm2/m128</td>
<td>Add packed doubleword integers from xmm2/m128 and xmm1.</td>
</tr>
</tbody>
</table>

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:

\[ \text{DEST}[7:0] \leftarrow \text{DEST}[7:0] + \text{SRC}[7:0]; \]
INSTRUCTION SET REFERENCE, N-Z

* 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 3rd 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 *;

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 3rd 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 ( __m128ia, __m128ib )
PADDD  __m64 _mm_add_pi32(__m64 m1, __m64 m2)
PADDW  __m128i _mm_add_epi32 ( __m128ia, __m128ib )

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.
INSTRUCTION SET REFERENCE, N-Z

#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

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:
\[ \text{DEST}[63-0] \leftarrow \text{DEST}[63-0] + \text{SRC}[63-0]; \]

PADDQ instruction with 128-Bit operands:
\[ \begin{align*}
\text{DEST}[63-0] & \leftarrow \text{DEST}[63-0] + \text{SRC}[63-0]; \\
\text{DEST}[127-64] & \leftarrow \text{DEST}[127-64] + \text{SRC}[127-64];
\end{align*} \]

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

**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]);
```

**Opcode Instruction Description**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F EC</td>
<td>PADDSB mm, mm/m64</td>
<td>Add packed signed byte integers from mm/m64 and mm and saturate the results.</td>
</tr>
<tr>
<td>66 0F EC</td>
<td>PADDSB xmm1, xmm1/m128</td>
<td>Add packed signed byte integers from xmm2/m128 and xmm1/m128 and saturate the results.</td>
</tr>
<tr>
<td>0F ED</td>
<td>PADDSW mm, mm/m64</td>
<td>Add packed signed word integers from mm/m64 and mm and saturate the results.</td>
</tr>
<tr>
<td>66 0F ED</td>
<td>PADDSW xmm1, xmm2/m128</td>
<td>Add packed signed word integers from xmm2/m128 and xmm1/m128 and saturate the results.</td>
</tr>
</tbody>
</table>
* 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 *;

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

**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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F DB /r</td>
<td>PAND mm, mm/m64</td>
<td>Bitwise AND mm/m64 and mm.</td>
</tr>
<tr>
<td>66 0F DB /r</td>
<td>PAND xmm1, xmm2/m128</td>
<td>Bitwise AND of xmm2/m128 and xmm1.</td>
</tr>
</tbody>
</table>

**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**

\[
\text{DEST} \leftarrow \text{DEST} \text{ AND } \text{SRC};
\]

**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 OSEFSR 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.
INSTRUCTION SET REFERENCE, N-Z

#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

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.
INSTRUCTION SET REFERENCE, N-Z

#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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 90</td>
<td>PAUSE</td>
<td>Gives hint to processor that improves performance of spin-wait loops.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F E0 /r</td>
<td>PAVGB mm1, mm2/m64</td>
<td>Average packed unsigned byte integers from mm2/m64 and mm1 with rounding.</td>
</tr>
<tr>
<td>66 0F E0, /r</td>
<td>PAVGB xmm1, xmm2/m128</td>
<td>Average packed unsigned byte integers from xmm2/m128 and xmm1 with rounding.</td>
</tr>
<tr>
<td>0F E3 /r</td>
<td>PAVGW mm1, mm2/m64</td>
<td>Average packed unsigned word integers from mm2/m64 and mm1 with rounding.</td>
</tr>
<tr>
<td>66 0F E3 /r</td>
<td>PAVGW xmm1, xmm2/m128</td>
<td>Average packed unsigned word integers from xmm2/m128 and xmm1 with rounding.</td>
</tr>
</tbody>
</table>

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:

\[
\text{SRC}(7-0) \leftarrow (\text{SRC}(7-0) + \text{DEST}(7-0) + 1) \gg 1; \quad \text{temp sum before shifting is 9 bits}^* \\
\text{SRC}(63-56) \leftarrow (\text{SRC}(63-56) + \text{DEST}(63-56) + 1) \gg 1;
\]

PAVGW instruction with 64-bit operands:

\[
\text{SRC}(15-0) \leftarrow (\text{SRC}(15-0) + \text{DEST}(15-0) + 1) \gg 1; \quad \text{temp sum before shifting is 17 bits}^* \\
\text{SRC}(63-48) \leftarrow (\text{SRC}(63-48) + \text{DEST}(63-48) + 1) \gg 1;
\]

PAVGB instruction with 128-bit operands:

\[
\text{SRC}(7-0) \leftarrow (\text{SRC}(7-0) + \text{DEST}(7-0) + 1) \gg 1; \quad \text{temp sum before shifting is 9 bits}^* \\
\text{SRC}(63-56) \leftarrow (\text{SRC}(63-56) + \text{DEST}(63-56) + 1) \gg 1;
\]

PAVGW instruction with 128-bit operands:

\[
\text{SRC}(15-0) \leftarrow (\text{SRC}(15-0) + \text{DEST}(15-0) + 1) \gg 1; \quad \text{temp sum before shifting is 17 bits}^* \\
\text{SRC}(127-48) \leftarrow (\text{SRC}(127-112) + \text{DEST}(127-112) + 1) \gg 1;
\]
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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 74 /r</td>
<td>PCMPEQB mm, mm/m64</td>
<td>Compare packed bytes in mm/m64 and mm for equality.</td>
</tr>
<tr>
<td>66 0F 74 /r</td>
<td>PCMPEQB xmm1, xmm2/m128</td>
<td>Compare packed bytes in xmm2/m128 and xmm1 for equality.</td>
</tr>
<tr>
<td>0F 75 /r</td>
<td>PCMPEQW mm, mm/m64</td>
<td>Compare packed words in mm/m64 and mm for equality.</td>
</tr>
<tr>
<td>66 0F 75 /r</td>
<td>PCMPEQW xmm1, xmm2/m128</td>
<td>Compare packed words in xmm2/m128 and xmm1 for equality.</td>
</tr>
<tr>
<td>0F 76 /r</td>
<td>PCMPEQD mm, mm/m64</td>
<td>Compare packed doublewords in mm/m64 and mm for equality.</td>
</tr>
<tr>
<td>66 0F 76 /r</td>
<td>PCMPEQD xmm1, xmm2/m128</td>
<td>Compare packed doublewords in xmm2/m128 and xmm1 for equality.</td>
</tr>
</tbody>
</table>

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 *
```

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 64 /r</td>
<td>PCMPGTB mm, mm/m64</td>
<td>Compare packed signed byte integers in mm and mm/m64 for greater than.</td>
</tr>
<tr>
<td>66 0F 64 /r</td>
<td>PCMPGTB xmm1, xmm2/m128</td>
<td>Compare packed signed byte integers in xmm1 and xmm2/m128 for greater than.</td>
</tr>
<tr>
<td>0F 65 /r</td>
<td>PCMPGTW mm, mm/m64</td>
<td>Compare packed signed word integers in mm and mm/m64 for greater than.</td>
</tr>
<tr>
<td>66 0F 65 /r</td>
<td>PCMPGTW xmm1, xmm2/m128</td>
<td>Compare packed signed word integers in xmm1 and xmm2/m128 for greater than.</td>
</tr>
<tr>
<td>0F 66 /r</td>
<td>PCMPGTD mm, mm/m64</td>
<td>Compare packed signed doubleword integers in mm and mm/m64 for greater than.</td>
</tr>
<tr>
<td>66 0F 66 /r</td>
<td>PCMPGTD xmm1, xmm2/m128</td>
<td>Compare packed signed doubleword integers in xmm1 and xmm2/m128 for greater than.</td>
</tr>
</tbody>
</table>

Description

Performs an SIMD signed compare for the greater value of the packed byte, word, or double-word 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;

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_cmpgt_pi16 (__m64 m1, __m64 m2)
DCMPGTD  __m64 __mm_cmpgt_pi32 (__m64 m1, __m64 m2)
PCMPGTB  __m128i __mm_cmpgt_epi8 ( __m128i a, __m128i b)
PCMPGTW  _m128i _mm_cmpgt_epi16 ( _m128i a, _m128i b
DCMPGT_D  _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. Execu-
tion 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. Execu-
tion 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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F C5 /r ib</td>
<td>PEXTRW r32, mm, imm8</td>
<td>Extract the word specified by imm8 from mm and move it to r32.</td>
</tr>
<tr>
<td>66 0F C5 /r ib</td>
<td>PEXTRW r32, xmm, imm8</td>
<td>Extract the word specified by imm8 from xmm and move it to a r32.</td>
</tr>
</tbody>
</table>

**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:

\[
\begin{align*}
& \text{SEL} \leftarrow \text{COUNT AND 3H;} \\
& \text{TEMP} \leftarrow (\text{SRC} >> (\text{SEL} \times 16)) \text{ AND FFFFH;} \\
& \text{r32}[15-0] \leftarrow \text{TEMP}[15-0]; \\
& \text{r32}[31-16] \leftarrow 0000H;
\end{align*}
\]

PEXTRW instruction with 128-bit source operand:

\[
\begin{align*}
& \text{SEL} \leftarrow \text{COUNT AND 7H;} \\
& \text{TEMP} \leftarrow (\text{SRC} >> (\text{SEL} \times 16)) \text{ AND FFFFH;} \\
& \text{r32}[15-0] \leftarrow \text{TEMP}[15-0]; \\
& \text{r32}[31-16] \leftarrow 0000H;
\end{align*}
\]

**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**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F C4 /r ib</td>
<td>PINSRW mm, r32/m16, imm8</td>
<td>Insert the low word from r32 or from m16 into mm at the word position specified by imm8.</td>
</tr>
<tr>
<td>66 0F C4 /r ib</td>
<td>PINSRW xmm, r32/m16, imm8</td>
<td>Move the low word of r32 or from m16 into xmm at the word position specified by imm8.</td>
</tr>
</tbody>
</table>

**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 ← 000000000000FFFF0000H;
  SEL ← 2: MASK ← 00000000FFFF00000000H;
  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 ← 0000000000000000000000000000FFFF0000H;
  SEL ← 2: MASK ← 000000000000000000000000FFFF00000000H;
  SEL ← 3: MASK ← 000000000000000000FFFF0000000000000000H;
  SEL ← 4: MASK ← 000000000000000000FFFF00000000000000000000H;
  SEL ← 5: MASK ← 000000000000000000FFFF0000000000000000000000H;
  SEL ← 6: MASK ← 000000000000000000FFFF000000000000000000000000H;
  SEL ← 7: MASK ← 000000000000000000FFFF0000000000000000000000000000H;
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)
INSTRUCTION SET REFERENCE, N-Z

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

**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.

**Operation**

PMADDWD instruction with 64-bit operands:

\[
\text{DEST}[31..0] \leftarrow (\text{DEST}[15..0] \times \text{SRC}[15..0]) + (\text{DEST}[31..16] \times \text{SRC}[31..16]); \\
\text{DEST}[63..32] \leftarrow (\text{DEST}[47..32] \times \text{SRC}[47..32]) + (\text{DEST}[63..48] \times \text{SRC}[63..48]);
\]
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 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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F EE /r</td>
<td>PMAXSW mm1, mm2/m64</td>
<td>Compare signed word integers in mm2/m64 and mm1 and return maximum values.</td>
</tr>
<tr>
<td>66 0F EE /r</td>
<td>PMAXSW xmm1, xmm2/m128</td>
<td>Compare signed word integers in xmm2/m128 and xmm1 and return maximum values.</td>
</tr>
</tbody>
</table>

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.
P.MAXUB—Maximum of Packed Unsigned Byte Integers

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F DE /r</td>
<td>P.MAXUB mm1, mm2/m64</td>
<td>Compare unsigned byte integers in mm2/m64 and mm1 and returns maximum values.</td>
</tr>
<tr>
<td>66 0F DE /r</td>
<td>P.MAXUB xmm1, xmm2/m128</td>
<td>Compare unsigned byte integers in xmm2/m128 and xmm1 and returns maximum values.</td>
</tr>
</tbody>
</table>

**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**

P.MAXUB 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

P.MAXUB 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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F EA /r</td>
<td>PMINSW mm1, mm2/m64</td>
<td>Compare signed word integers in mm2/m64 and mm1 and return minimum values.</td>
</tr>
<tr>
<td>66 0F EA /r</td>
<td>PMINSW xmm1, xmm2/m128</td>
<td>Compare signed word integers in xmm2/m128 and xmm1 and return minimum values.</td>
</tr>
</tbody>
</table>

**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

#### 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

#### Opcode Instruction Description

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F DA /r</td>
<td>PMINUB mm1, mm2/m64</td>
<td>Compare unsigned byte integers in mm2/m64 and mm1 and return minimum values.</td>
</tr>
<tr>
<td>66 0F DA /r</td>
<td>PMINUB xmm1, xmm2/m128</td>
<td>Compare unsigned byte integers in xmm2/m128 and xmm1 and return minimum values.</td>
</tr>
</tbody>
</table>
INSTRUCTION SET REFERENCE, N-Z

Intel C/C++ Compiler Intrinsic Equivalent

PMINUB __m64 __m_min_pu8 (__m64 a, __m64 b)
PMINUB __m128i __m_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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F D7 /r</td>
<td>PMOVMSKB r32, mm</td>
<td>Move a byte mask of mm to r32.</td>
</tr>
<tr>
<td>66 0F D7 /r</td>
<td>PMOVMSKB r32, xmm</td>
<td>Move a byte mask of xmm to r32.</td>
</tr>
</tbody>
</table>

**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:
\[ r32[0] \leftarrow \text{SRC}[7]; \]
\[ r32[1] \leftarrow \text{SRC}[15]; \]
* repeat operation for bytes 2 through 6;
\[ r32[7] \leftarrow \text{SRC}[63]; \]
\[ r32[31-8] \leftarrow 000000H; \]

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

**Intel C/C++ Compiler Intrinsic Equivalent**

PMOVMSKB \hspace{1em} \text{int\_mm\_movemask\_pi8(\_m64 a)}

PMOVMSKB \hspace{1em} \text{int\_mm\_movemask\_epi8 (\_m128i a)}

**Flags Affected**

None.

**Protected Mode Exceptions**

#UD  \hspace{1em} 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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F E4 /r</td>
<td>PMULHUW mm1, mm2/m64</td>
<td>Multiply the packed unsigned word integers in mm1 register and mm2/m64, and store the high 16 bits of the results in mm1.</td>
</tr>
<tr>
<td>66 0F E4 /r</td>
<td>PMULHUW xmm1, xmm2/m128</td>
<td>Multiply the packed unsigned word integers in xmm1 and xmm2/m128, and store the high 16 bits of the results in xmm1.</td>
</tr>
</tbody>
</table>

**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.

**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:

\[
\begin{align*}
\text{TEMP0}[31-0] & \leftarrow \text{DEST}[15-0] \times \text{SRC}[15-0]; \quad \text{Unsigned multiplication} \\
\text{TEMP1}[31-0] & \leftarrow \text{DEST}[31-16] \times \text{SRC}[31-16]; \\
\text{TEMP2}[31-0] & \leftarrow \text{DEST}[47-32] \times \text{SRC}[47-32]; \\
\text{TEMP3}[31-0] & \leftarrow \text{DEST}[63-48] \times \text{SRC}[63-48]; \\
\text{TEMP4}[31-0] & \leftarrow \text{DEST}[79-64] \times \text{SRC}[79-64]; \\
\text{TEMP5}[31-0] & \leftarrow \text{DEST}[95-80] \times \text{SRC}[95-80]; \\
\text{TEMP6}[31-0] & \leftarrow \text{DEST}[111-96] \times \text{SRC}[111-96]; \\
\text{TEMP7}[31-0] & \leftarrow \text{DEST}[127-112] \times \text{SRC}[127-112]; \\
\text{DEST}[15-0] & \leftarrow \text{TEMP0}[31-16]; \\
\text{DEST}[31-16] & \leftarrow \text{TEMP1}[31-16]; \\
\text{DEST}[47-32] & \leftarrow \text{TEMP2}[31-16]; \\
\text{DEST}[63-48] & \leftarrow \text{TEMP3}[31-16]; \\
\text{DEST}[79-64] & \leftarrow \text{TEMP4}[31-16]; \\
\text{DEST}[95-80] & \leftarrow \text{TEMP5}[31-16]; \\
\text{DEST}[111-96] & \leftarrow \text{TEMP6}[31-16]; \\
\text{DEST}[127-112] & \leftarrow \text{TEMP7}[31-16];
\end{align*}
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

- PMULHUW \(_{m64} \text{__m64}_\text{mulhi}_\text{pu16}(\_m64 \text{a, } \_m64 \text{b})\)
- PMULHUW \(_{m128i} \text{__m128i}_\text{mulhi}_\text{epu16}(\_m128i \text{a, } \_m128i \text{b})\)

**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

### 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:
1. $\text{TEMP0}[31-0] \leftarrow \text{DEST}[15-0] \cdot \text{SRC}[15-0]$; * Signed multiplication *
2. $\text{TEMP1}[31-0] \leftarrow \text{DEST}[31-16] \cdot \text{SRC}[31-16]$;
3. $\text{TEMP2}[31-0] \leftarrow \text{DEST}[47-32] \cdot \text{SRC}[47-32]$;
4. $\text{TEMP3}[31-0] \leftarrow \text{DEST}[63-48] \cdot \text{SRC}[63-48]$;
5. $\text{DEST}[15-0] \leftarrow \text{TEMP0}[31-16]$;
6. $\text{DEST}[31-16] \leftarrow \text{TEMP1}[31-16]$;
7. $\text{DEST}[47-32] \leftarrow \text{TEMP2}[31-16]$;
8. $\text{DEST}[63-48] \leftarrow \text{TEMP3}[31-16]$;

PMULHW instruction with 128-bit operands:
1. $\text{TEMP0}[31-0] \leftarrow \text{DEST}[15-0] \cdot \text{SRC}[15-0]$; * Signed multiplication *
2. $\text{TEMP1}[31-0] \leftarrow \text{DEST}[31-16] \cdot \text{SRC}[31-16]$;
3. $\text{TEMP2}[31-0] \leftarrow \text{DEST}[47-32] \cdot \text{SRC}[47-32]$;
4. $\text{TEMP3}[31-0] \leftarrow \text{DEST}[63-48] \cdot \text{SRC}[63-48]$;
5. $\text{TEMP4}[31-0] \leftarrow \text{DEST}[79-64] \cdot \text{SRC}[79-64]$;
6. $\text{TEMP5}[31-0] \leftarrow \text{DEST}[95-80] \cdot \text{SRC}[95-80]$;
7. $\text{TEMP6}[31-0] \leftarrow \text{DEST}[111-96] \cdot \text{SRC}[111-96]$;
8. $\text{TEMP7}[31-0] \leftarrow \text{DEST}[127-112] \cdot \text{SRC}[127-112]$;
9. $\text{DEST}[15-0] \leftarrow \text{TEMP0}[31-16]$;
10. $\text{DEST}[31-16] \leftarrow \text{TEMP1}[31-16]$;
11. $\text{DEST}[47-32] \leftarrow \text{TEMP2}[31-16]$;
12. $\text{DEST}[63-48] \leftarrow \text{TEMP3}[31-16]$;
13. $\text{DEST}[79-64] \leftarrow \text{TEMP4}[31-16]$;
14. $\text{DEST}[95-80] \leftarrow \text{TEMP5}[31-16]$;
15. $\text{DEST}[111-96] \leftarrow \text{TEMP6}[31-16]$;
16. $\text{DEST}[127-112] \leftarrow \text{TEMP7}[31-16]$;

### 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.
INSTRUCTION SET REFERENCE, N-Z

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

**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.

**Operation**

PMULLW instruction with 64-bit operands:

\[
\begin{align*}
\text{TEMP0}[31-0] & \leftarrow \text{DEST}[15-0] \times \text{SRC}[15-0]; \quad \text{Signed multiplication} \\
\text{TEMP1}[31-0] & \leftarrow \text{DEST}[31-16] \times \text{SRC}[31-16]; \\
\text{TEMP2}[31-0] & \leftarrow \text{DEST}[47-32] \times \text{SRC}[47-32]; \\
\text{TEMP3}[31-0] & \leftarrow \text{DEST}[63-48] \times \text{SRC}[63-48]; \\
\text{DEST}[15-0] & \leftarrow \text{TEMP0}[15-0]; \\
\text{DEST}[31-16] & \leftarrow \text{TEMP1}[15-0]; \\
\text{DEST}[47-32] & \leftarrow \text{TEMP2}[15-0]; \\
\text{DEST}[63-48] & \leftarrow \text{TEMP3}[15-0];
\end{align*}
\]

PMULLW instruction with 64-bit operands:

\[
\begin{align*}
\text{TEMP0}[31-0] & \leftarrow \text{DEST}[15-0] \times \text{SRC}[15-0]; \quad \text{Signed multiplication} \\
\end{align*}
\]
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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F F4 /r</td>
<td>PMULUDQ mm1, mm2/m64</td>
<td>Multiply unsigned doubleword integer in mm1 by unsigned doubleword integer in mm2/m64, and store the quadword result in mm1.</td>
</tr>
<tr>
<td>66 0F F4 /r</td>
<td>PMULUDQ xmm1, xmm2/m128</td>
<td>Multiply packed unsigned doubleword integers in xmm1 by packed unsigned doubleword integers in xmm2/m128, and store the quadword results in xmm1.</td>
</tr>
</tbody>
</table>

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

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

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 interrupt¹. 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;
   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

**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**

```plaintext
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;
```

**Opcode Instruction Description**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>POPA</td>
<td>Pop DI, SI, BP, BX, DX, CX, and AX.</td>
</tr>
<tr>
<td>61</td>
<td>POPAD</td>
<td>Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9D</td>
<td>POPF</td>
<td>Pop top of stack into lower 16 bits of EFLAGS.</td>
</tr>
<tr>
<td>9D</td>
<td>POPFD</td>
<td>Pop top of stack into EFLAGS.</td>
</tr>
</tbody>
</table>

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

\[
\text{IF VM=0 (* Not in Virtual-8086 Mode *)} \\
\quad \text{THEN IF CPL=0} \\
\quad \quad \text{THEN} \\
\quad \quad \quad \text{IF OperandSize = 32;} \\
\quad \quad \quad \text{THEN}
\]
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

**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**

\[
\text{DEST} \leftarrow \text{DEST} \text{ OR } \text{SRC};
\]

**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.
PREFETCHh—Prefetch Data Into Caches

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 18 /1</td>
<td>PREFETCHT0 m8</td>
<td>Move data from m8 closer to the processor using T0 hint.</td>
</tr>
<tr>
<td>0F 18 /2</td>
<td>PREFETCHT1 m8</td>
<td>Move data from m8 closer to the processor using T1 hint.</td>
</tr>
<tr>
<td>0F 18 /3</td>
<td>PREFETCHT2 m8</td>
<td>Move data from m8 closer to the processor using T2 hint.</td>
</tr>
<tr>
<td>0F 18 /0</td>
<td>PREFETCHNTA m8</td>
<td>Move data from m8 closer to the processor using NTA hint.</td>
</tr>
</tbody>
</table>

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 PREFETCHh 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\textit{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\textit{h} instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCH\textit{h} instruction is also unordered with respect to CLFLUSH instructions, other PREFETCH\textit{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**

\texttt{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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F F6 /r</td>
<td>PSADBW mm1, mm2/m64</td>
<td>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.</td>
</tr>
<tr>
<td>66 0F F6 /r</td>
<td>PSADBW xmm1, xmm2/m128</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.
Operation

PSADBW instructions when using 64-bit operands:

\[
\text{TEMP}_0 \leftarrow \text{ABS}(\text{DEST}[7-0] - \text{SRC}[7-0]);
\]

* repeat operation for bytes 2 through 6 *;

\[
\text{TEMP}_7 \leftarrow \text{ABS}(\text{DEST}[63-56] - \text{SRC}[63-56]);
\]

\[
\text{DEST}[15:0] \leftarrow \text{SUM}(\text{TEMP}_0...\text{TEMP}_7);
\]

\[
\text{DEST}[63:16] \leftarrow 000000000000H;
\]

PSADBW instructions when using 128-bit operands:

\[
\text{TEMP}_0 \leftarrow \text{ABS}(\text{DEST}[7-0] - \text{SRC}[7-0]);
\]

* repeat operation for bytes 2 through 14 *;

\[
\text{TEMP}_{15} \leftarrow \text{ABS}(\text{DEST}[127-120] - \text{SRC}[127-120]);
\]

\[
\text{DEST}[15-0] \leftarrow \text{SUM}(\text{TEMP}_0...\text{TEMP}_7);
\]

\[
\text{DEST}[63-6] \leftarrow 000000000000H;
\]

\[
\text{DEST}[79-64] \leftarrow \text{SUM}(\text{TEMP}_8...\text{TEMP}_{15});
\]

\[
\text{DEST}[127-80] \leftarrow 000000000000H;
\]

Intel C/C++ Compiler Intrinsic Equivalent

\[
\text{PSADBW } \_\_m64\_\_mm\_sad\_pu8 (\_\_m64\ a, \_\_m64\ b)
\]

\[
\text{PSADBW } \_\_m128i\_\_mm\_sad\_epu8 (\_\_m128i\ a, \_\_m128i\ b)
\]

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.
INSTRUCTION SET REFERENCE, N-Z

PSHUFD—Shuffle Packed Doublewords

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 70</td>
<td>PSHUFD xmm1, xmm2/m128, imm8</td>
<td>Shuffle the doublewords in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.</td>
</tr>
</tbody>
</table>

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.

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

\[
\begin{align*}
\text{DEST}[31-0] & \leftarrow (\text{SRC} >> (\text{ORDER}[1-0] \times 32))[31-0] \\
\text{DEST}[63-32] & \leftarrow (\text{SRC} >> (\text{ORDER}[3-2] \times 32))[31-0] \\
\text{DEST}[95-64] & \leftarrow (\text{SRC} >> (\text{ORDER}[5-4] \times 32))[31-0] \\
\text{DEST}[127-96] & \leftarrow (\text{SRC} >> (\text{ORDER}[7-6] \times 32))[31-0]
\end{align*}
\]
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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 70</td>
<td>PSHUFHW xmm1, xmm2/m128, imm8</td>
<td>Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.</td>
</tr>
</tbody>
</table>

**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**

\[
\begin{align*}
\text{DEST}[63-0] & \leftarrow \text{SRC}[63-0] \\
\text{DEST}[79-64] & \leftarrow (\text{SRC} \gg (\text{ORDER}[1-0] \times 16))[79-64] \\
\text{DEST}[95-80] & \leftarrow (\text{SRC} \gg (\text{ORDER}[3-2] \times 16))[79-64] \\
\text{DEST}[111-96] & \leftarrow (\text{SRC} \gg (\text{ORDER}[5-4] \times 16))[79-64] \\
\text{DEST}[127-112] & \leftarrow (\text{SRC} \gg (\text{ORDER}[7-6] \times 16))[79-64]
\end{align*}
\]

**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**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 70  lr lb</td>
<td>PSHUFLW xmm1, xmm2/m128, imm8</td>
<td>Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.</td>
</tr>
</tbody>
</table>

**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**

\[
\begin{align*}
\text{DEST}[15-0] & \leftarrow (\text{SRC} >> (\text{ORDER}[1-0] \times 16)) [15-0] \\
\text{DEST}[31-16] & \leftarrow (\text{SRC} >> (\text{ORDER}[3-2] \times 16)) [15-0] \\
\text{DEST}[47-32] & \leftarrow (\text{SRC} >> (\text{ORDER}[5-4] \times 16)) [15-0] \\
\text{DEST}[63-48] & \leftarrow (\text{SRC} >> (\text{ORDER}[7-6] \times 16)) [15-0] \\
\text{DEST}[127-64] & \leftarrow (\text{SRC}[127-64])
\end{align*}
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

`PSHUFLW _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.
INSTRUCTION SET REFERENCE, N-Z

#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

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

\[
\begin{align*}
\text{DEST}[15-0] &\leftarrow (\text{SRC} >> (\text{ORDER}[1-0] \times 16))[15-0] \\
\text{DEST}[31-16] &\leftarrow (\text{SRC} >> (\text{ORDER}[3-2] \times 16))[15-0] \\
\text{DEST}[47-32] &\leftarrow (\text{SRC} >> (\text{ORDER}[5-4] \times 16))[15-0] \\
\text{DEST}[63-48] &\leftarrow (\text{SRC} >> (\text{ORDER}[7-6] \times 16))[15-0]
\end{align*}
\]

Intel C/C++ Compiler Intrinsic Equivalent

\[
\text{PSHUFW} \quad \_\_\text{m64} \_\_\text{mm}_\text{shuffle}_\text{pi16}(\_\_\text{m64} \text{a, int n})
\]

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 73 /7 ib</td>
<td>PSLLDQ xmm1, imm8</td>
<td>Shift xmm1 left by imm8 bytes while shifting in 0s.</td>
</tr>
</tbody>
</table>

**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**

\[ \text{TEMP} \leftarrow \text{COUNT}; \]
\[ \text{if (TEMP} > 15) \text{TEMP} \leftarrow 16; \]
\[ \text{DEST} \leftarrow \text{DEST} \ll (\text{TEMP} \times 8); \]

**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

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

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](image-url)
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 double-words in the destination operand; and the PSLLQ instruction shifts the quadword (or quad-words) in the destination operand.

**Operation**

**PSLLW instruction with 64-bit operand:**
\[
\text{IF (COUNT} > 15) \\
\text{THEN} \\
\quad \text{DEST}[64..0] \leftarrow 0000000000000000H \\
\text{ELSE} \\
\quad \text{DEST}[15..0] \leftarrow \text{ZeroExtend}(\text{DEST}[15..0] \ll \text{COUNT}); \\
\quad * \text{repeat shift operation for 2nd and 3rd words *}; \\
\quad \text{DEST}[63..48] \leftarrow \text{ZeroExtend}(\text{DEST}[63..48] \ll \text{COUNT}); \\
\FI;
\]

**PSLLD instruction with 64-bit operand:**
\[
\text{IF (COUNT} > 31) \\
\text{THEN} \\
\quad \text{DEST}[64..0] \leftarrow 0000000000000000H \\
\text{ELSE} \\
\quad \text{DEST}[31..0] \leftarrow \text{ZeroExtend}(\text{DEST}[31..0] \ll \text{COUNT}); \\
\quad \text{DEST}[63..32] \leftarrow \text{ZeroExtend}(\text{DEST}[63..32] \ll \text{COUNT}); \\
\FI;
\]

**PSLLQ instruction with 64-bit operand:**
\[
\text{IF (COUNT} > 63) \\
\text{THEN} \\
\quad \text{DEST}[64..0] \leftarrow 0000000000000000H \\
\text{ELSE} \\
\quad \text{DEST} \leftarrow \text{ZeroExtend}(\text{DEST} \ll \text{COUNT}); \\
\FI;
\]

**PSLLW instruction with 128-bit operand:**
\[
\text{IF (COUNT} > 15) \\
\text{THEN} \\
\quad \text{DEST}[128..0] \leftarrow 000000000000000000000000000000000000000000H \\
\text{ELSE} \\
\quad \text{DEST}[15-0] \leftarrow \text{ZeroExtend}(\text{DEST}[15-0] \ll \text{COUNT}); \\
\quad * \text{repeat shift operation for 2nd through 7th words *}; \\
\quad \text{DEST}[127-112] \leftarrow \text{ZeroExtend}(\text{DEST}[127-112] \ll \text{COUNT}); \\
\FI;
\]

**PSLLD instruction with 128-bit operand:**
\[
\text{IF (COUNT} > 31) \\
\text{THEN} \\
\quad \text{DEST}[128..0] \leftarrow 000000000000000000000000000000000000000000H \\
\text{ELSE} \\
\quad \text{DEST}[31-0] \leftarrow \text{ZeroExtend}(\text{DEST}[31-0] \ll \text{COUNT}); \\
\quad * \text{repeat shift operation for 2nd through 7th words *}; \\
\quad \text{DEST}[63-32] \leftarrow \text{ZeroExtend}(\text{DEST}[63-32] \ll \text{COUNT}); \\
\FI;
\]
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_sll_pi16(__m128i m, __m128i count)
PSLLD __m64 _mm_slli_epi32(__m64 m, int count)
PSLLD __m64 _mm_sll_epi32(__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**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F E1 /r</td>
<td>PSRAW mm, mm/m64</td>
<td>Shift words in mm right by mm/m64 while shifting in sign bits.</td>
</tr>
<tr>
<td>66 0F E1 /r</td>
<td>PSRAW xmm1, xmm2/m128</td>
<td>Shift words in xmm1 right by xmm2/m128 while shifting in sign bits.</td>
</tr>
<tr>
<td>0F 71 /4 ib</td>
<td>PSRAW mm, imm8</td>
<td>Shift words in mm right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>66 0F 71 /4 ib</td>
<td>PSRAW xmm1, imm8</td>
<td>Shift words in xmm1 right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>0F E2 /r</td>
<td>PSRAD mm, mm/m64</td>
<td>Shift doublewords in mm right by mm/m64 while shifting in sign bits.</td>
</tr>
<tr>
<td>66 0F E2 /r</td>
<td>PSRAD xmm1, xmm2/m128</td>
<td>Shift doubleword in xmm1 right by xmm2/m128 while shifting in sign bits.</td>
</tr>
<tr>
<td>0F 72 /4 ib</td>
<td>PSRAD mm, imm8</td>
<td>Shift doublewords in mm right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>66 0F 72 /4 ib</td>
<td>PSRAD xmm1, imm8</td>
<td>Shift doublewords in xmm1 right by imm8 while shifting in sign bits.</td>
</tr>
</tbody>
</table>

**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](image)

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 73</td>
<td>PSRLDQ xmm1, imm8</td>
<td>Shift xmm1 right by imm8 while shifting in 0s.</td>
</tr>
</tbody>
</table>

**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

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

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](image-url)
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 double-words in the destination operand; and the PSRLQ instruction shifts the quadword (or quad-words) 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

**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.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F F8 /r</td>
<td>PSUBB mm, mm/m64</td>
<td>Subtract packed byte integers in mm/m64 from packed byte integers in mm.</td>
</tr>
<tr>
<td>66 0F F8 /r</td>
<td>PSUBB xmm1, xmm2/m128</td>
<td>Subtract packed byte integers in xmm2/m128 from packed byte integers in xmm1.</td>
</tr>
<tr>
<td>0F F9 /r</td>
<td>PSUBW mm, mm/m64</td>
<td>Subtract packed word integers in mm/m64 from packed word integers in mm.</td>
</tr>
<tr>
<td>66 0F F9 /r</td>
<td>PSUBW xmm1, xmm2/m128</td>
<td>Subtract packed word integers in xmm2/m128 from packed word integers in xmm1.</td>
</tr>
<tr>
<td>0F FA /r</td>
<td>PSUBD mm, mm/m64</td>
<td>Subtract packed doubleword integers in mm/m64 from packed doubleword integers in mm.</td>
</tr>
<tr>
<td>66 0F FA /r</td>
<td>PSUBD xmm1, xmm2/m128</td>
<td>Subtract packed doubleword integers in xmm2/mem128 from packed doubleword integers in xmm1.</td>
</tr>
</tbody>
</table>
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 *;

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

PSUDB instruction with 128-bit operands:
   DEST[31-0] ← DEST[31-0] – SRC[31-0];
   * repeat subtract operation for 2nd and 3rd doubleword *;

Intel C/C++ Compiler Intrinsic Equivalents

PSUDB     _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)
PSUDB     __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.
INSTRUCTION SET REFERENCE, N-Z

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

### 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:**

\[
\text{DEST}[63-0] \leftarrow \text{DEST}[63-0] - \text{SRC}[63-0];
\]

**PSUBQ instruction with 128-Bit operands:**

\[
\text{DEST}[63-0] \leftarrow \text{DEST}[63-0] - \text{SRC}[63-0];
\]
\[
\text{DEST}[127-64] \leftarrow \text{DEST}[127-64] - \text{SRC}[127-64];
\]

### 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

**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:

\[
\text{DEST}[7..0] \leftarrow \text{SaturateToSignedByte}(\text{DEST}[7..0] - \text{SRC}[7..0]);
\]

* repeat subtract operation for 2nd through 7th bytes *

\[
\text{DEST}[63..56] \leftarrow \text{SaturateToSignedByte}(\text{DEST}[63..56] - \text{SRC}[63..56]);
\]

PSUBSB instruction with 128-bit operands:

\[
\text{DEST}[7-0] \leftarrow \text{SaturateToSignedByte}(\text{DEST}[7-0] - \text{SRC}[7-0]);
\]

* repeat subtract operation for 2nd through 14th bytes *

\[
\text{DEST}[127-120] \leftarrow \text{SaturateToSignedByte}(\text{DEST}[111-120] - \text{SRC}[127-120]);
\]

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F E8 /r</td>
<td>PSUBSB mm, mm/m64</td>
<td>Subtract signed packed bytes in mm/m64 from signed packed bytes in mm and saturate results.</td>
</tr>
<tr>
<td>66 0F E8 /r</td>
<td>PSUBSB xmm1, xmm2/m128</td>
<td>Subtract packed signed byte integers in xmm2/m128 from packed signed byte integers in xmm1 and saturate results.</td>
</tr>
<tr>
<td>0F E9 /r</td>
<td>PSUBSW mm, mm/m64</td>
<td>Subtract signed packed words in mm/m64 from signed packed words in mm and saturate results.</td>
</tr>
<tr>
<td>66 0F E9 /r</td>
<td>PSUBSW xmm1, xmm2/m128</td>
<td>Subtract packed signed word integers in xmm2/m128 from packed signed word integers in xmm1 and saturate results.</td>
</tr>
</tbody>
</table>
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
* repeat subtract operation for 2nd through 7th words *;

Intel C/C++ Compiler Intrinsic Equivalents
PSUBSB     _mm_subs_pi8(__m64 m1, __m64 m2)
PSUBSB     _mm_subs_epi8(__m128i m1, __m128i m2)
PSUBSW     _mm_subs_pi16(__m64 m1, __m64 m2)
PSUBSW     _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

**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:**

\[
\text{DEST}[7..0] \leftarrow \text{SaturateToUnsignedByte}(\text{DEST}[7..0] - \text{SRC}[7..0]);
\]

* repeat add operation for 2nd through 7th bytes *:

\[
\text{DEST}[63..56] \leftarrow \text{SaturateToUnsignedByte}(\text{DEST}[63..56] - \text{SRC}[63..56])
\]

**PSUBUSB instruction with 128-bit operands:**

\[
\text{DEST}[7..0] \leftarrow \text{SaturateToUnsignedByte}(\text{DEST}[7..0] - \text{SRC}[7..0]);
\]

* repeat add operation for 2nd through 14th bytes *:

\[
\text{DEST}[127..120] \leftarrow \text{SaturateToUnsignedByte}(\text{DEST}[127..120] - \text{SRC}[127..120]);
\]

**PSUBUSW instruction with 64-bit operands:**

\[
\text{DEST}[15..0] \leftarrow \text{SaturateToUnsignedWord}(\text{DEST}[15..0] - \text{SRC}[15..0]);
\]
* 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 *:

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.
INSTRUCTION SET REFERENCE, N-Z

#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

**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.

```
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 68</td>
<td>PUNPCKHBW mm, mm/m64</td>
<td>Unpack and interleave high-order bytes from mm and mm/m64 into mm.</td>
</tr>
<tr>
<td>66 0F 68</td>
<td>PUNPCKHBW xmm1, xmm2/m128</td>
<td>Unpack and interleave high-order bytes from xmm1 and xmm2/m128 into xmm1.</td>
</tr>
<tr>
<td>0F 69</td>
<td>PUNPCKHWD mm, mm/m64</td>
<td>Unpack and interleave high-order words from mm and mm/m64 into mm.</td>
</tr>
<tr>
<td>66 0F 69</td>
<td>PUNPCKHWD xmm1, xmm2/m128</td>
<td>Unpack and interleave high-order words from xmm1 and xmm2/m128 into xmm1.</td>
</tr>
<tr>
<td>0F 6A</td>
<td>PUNPCKHDQ mm, mm/m64</td>
<td>Unpack and interleave high-order doublewords from mm and mm/m64 into mm.</td>
</tr>
<tr>
<td>66 0F 6A</td>
<td>PUNPCKHDQ xmm1, xmm2/m128</td>
<td>Unpack and interleave high-order doublewords from xmm1 and xmm2/m128 into xmm1.</td>
</tr>
<tr>
<td>0F 6D</td>
<td>PUNPCKHQDQ xmm1, xmm2/m128</td>
<td>Unpack and interleave high-order quadwords from xmm1 and xmm2/m128 into xmm1.</td>
</tr>
<tr>
<td>66 0F 6D</td>
<td>PUNPCKHQDQ xmm1, xmm2/m128</td>
<td>Unpack and interleave high-order quadwords from xmm1 and xmm2/m128 into xmm1.</td>
</tr>
</tbody>
</table>
```

**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];

PUNPCKHWD 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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 60 /r</td>
<td>PUNPCKLBW mm, mm/m32</td>
<td>Interleave low-order bytes from mm and mm/m32 into mm.</td>
</tr>
<tr>
<td>66 0F 60 /r</td>
<td>PUNPCKLBW xmm1, xmm2/m128</td>
<td>Interleave low-order bytes from xmm1 and xmm2/m128 into xmm1.</td>
</tr>
<tr>
<td>0F 61 /r</td>
<td>PUNPCKLWD mm, mm/m32</td>
<td>Interleave low-order words from mm and mm/m32 into mm.</td>
</tr>
<tr>
<td>66 0F 61 /r</td>
<td>PUNPCKLWD xmm1, xmm2/m128</td>
<td>Interleave low-order words from xmm1 and xmm2/m128 into xmm1.</td>
</tr>
<tr>
<td>0F 62 /r</td>
<td>PUNPCKLDQ mm, mm/m32</td>
<td>Interleave low-order doublewords from mm and mm/m32 into mm.</td>
</tr>
<tr>
<td>66 0F 62 /r</td>
<td>PUNPCKLDQ xmm1, xmm2/m128</td>
<td>Interleave low-order doublewords from xmm1 and xmm2/m128 into xmm1.</td>
</tr>
<tr>
<td>66 0F 6C /r</td>
<td>PUNPCKLQDQ xmm1, xmm2/m128</td>
<td>Interleave low-order quadwords from xmm1 and xmm2/m128 into xmm1.</td>
</tr>
</tbody>
</table>

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.

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 inter-leaves 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:**
- $\text{DEST}[63..56] \leftarrow \text{SRC}[31..24];$
- $\text{DEST}[55..48] \leftarrow \text{DEST}[31..24];$
- $\text{DEST}[47..40] \leftarrow \text{SRC}[23..16];$
- $\text{DEST}[39..32] \leftarrow \text{DEST}[23..16];$
- $\text{DEST}[31..24] \leftarrow \text{SRC}[15..8];$
- $\text{DEST}[23..16] \leftarrow \text{DEST}[15..8];$
- $\text{DEST}[15..8] \leftarrow \text{SRC}[7..0];$
- $\text{DEST}[7..0] \leftarrow \text{DEST}[7..0];$

**PUNPCKLWD instruction with 64-bit operands:**
- $\text{DEST}[63..48] \leftarrow \text{SRC}[31..16];$
- $\text{DEST}[47..32] \leftarrow \text{DEST}[31..16];$
- $\text{DEST}[31..16] \leftarrow \text{SRC}[15..0];$
- $\text{DEST}[15..0] \leftarrow \text{DEST}[15..0];$

**PUNPCKLDQ instruction with 64-bit operands:**
- $\text{DEST}[63..32] \leftarrow \text{SRC}[31..0];$
- $\text{DEST}[31..0] \leftarrow \text{DEST}[31..0];$

**PUNPCKLBW instruction with 128-bit operands:**
- $\text{DEST}[7-0] \leftarrow \text{DEST}[7-0];$
- $\text{DEST}[15-8] \leftarrow \text{SRC}[7-0];$
- $\text{DEST}[23-16] \leftarrow \text{DEST}[15-8];$
- $\text{DEST}[31-24] \leftarrow \text{SRC}[15-8];$
- $\text{DEST}[39-32] \leftarrow \text{DEST}[23-16];$
- $\text{DEST}[47-40] \leftarrow \text{SRC}[23-16];$
- $\text{DEST}[55-48] \leftarrow \text{DEST}[31-24];$
- $\text{DEST}[63-56] \leftarrow \text{SRC}[31-24];$
- $\text{DEST}[71-64] \leftarrow \text{DEST}[39-32];$
- $\text{DEST}[79-72] \leftarrow \text{SRC}[39-32];$
- $\text{DEST}[87-80] \leftarrow \text{DEST}[47-40];$
- $\text{DEST}[95-88] \leftarrow \text{SRC}[47-40];$
- $\text{DEST}[103-96] \leftarrow \text{DEST}[55-48];$
- $\text{DEST}[111-104] \leftarrow \text{SRC}[55-48];$
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];

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF /6</td>
<td>PUSH r/m16</td>
<td>Push r/m16.</td>
</tr>
<tr>
<td>FF /6</td>
<td>PUSH r/m32</td>
<td>Push r/m32.</td>
</tr>
<tr>
<td>50+rw</td>
<td>PUSH r16</td>
<td>Push r16.</td>
</tr>
<tr>
<td>50+rd</td>
<td>PUSH r32</td>
<td>Push r32.</td>
</tr>
<tr>
<td>68</td>
<td>PUSH imm16</td>
<td>Push imm16.</td>
</tr>
<tr>
<td>68</td>
<td>PUSH imm32</td>
<td>Push imm32.</td>
</tr>
<tr>
<td>0E</td>
<td>PUSH CS</td>
<td>Push CS.</td>
</tr>
<tr>
<td>16</td>
<td>PUSH SS</td>
<td>Push SS.</td>
</tr>
<tr>
<td>1E</td>
<td>PUSH DS</td>
<td>Push DS.</td>
</tr>
<tr>
<td>06</td>
<td>PUSH ES</td>
<td>Push ES.</td>
</tr>
<tr>
<td>0F A0</td>
<td>PUSH FS</td>
<td>Push FS.</td>
</tr>
<tr>
<td>0F A8</td>
<td>PUSH GS</td>
<td>Push GS.</td>
</tr>
</tbody>
</table>

Description

Decrement 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

**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);

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>PUSHA</td>
<td>Push AX, CX, DX, BX, original SP, BP, SI, and DI.</td>
</tr>
<tr>
<td>60</td>
<td>PUSHAD</td>
<td>Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9C</td>
<td>PUSHF</td>
<td>Push lower 16 bits of EFLAGS.</td>
</tr>
<tr>
<td>9C</td>
<td>PUSHFD</td>
<td>Push EFLAGS.</td>
</tr>
</tbody>
</table>

**Description**

Decrement 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**

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

4-146  Vol. 2B
ELSE (* In Virtual-8086 Mode with IOPL less than 3 *)
  #GP(0); (* Trap to virtual-8086 monitor *)
Fl;

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.
INSTRUCTION SET REFERENCE, N-Z

PXOR—Logical Exclusive OR

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F EF /r</td>
<td>PXOR mm, mm/m64</td>
<td>Bitwise XOR of mm/m64 and mm.</td>
</tr>
<tr>
<td>66 0F EF /r</td>
<td>PXOR xmm1, xmm2/m128</td>
<td>Bitwise XOR of xmm2/m128 and xmm1.</td>
</tr>
</tbody>
</table>

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.
INSTRUCTION SET REFERENCE, N-Z

#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

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

**Description**
Perform 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:

\[ |\text{Relative Error}| \leq 1.5 \times 2^{-12} \]

The RCPPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an \( \infty \) 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 \( 1.11111111110100000000000B \times 2^{125} \) are guaranteed to not produce tiny results; input values less than or equal to \( 1.00000000000110000000001B \times 2^{126} \) are guaranteed to produce tiny results, which in turn are 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**

\[
\begin{align*}
\text{DEST}[31-0] & \leftarrow \text{APPROXIMATE}(1.0/\text{SRC}[31-0]); \\
\text{DEST}[63-32] & \leftarrow \text{APPROXIMATE}(1.0/\text{SRC}[63-32]); \\
\text{DEST}[95-64] & \leftarrow \text{APPROXIMATE}(1.0/\text{SRC}[95-64]); \\
\text{DEST}[127-96] & \leftarrow \text{APPROXIMATE}(1.0/\text{SRC}[127-96]);
\end{align*}
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

\[
\text{RCCPS} \quad \text{__m128} \quad \text{mm_rcp_ps(__m128 a)}
\]

**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

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:

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

The RCPSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ 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 \[1.11111111101000000000B \times 2^{125}\] are guaranteed to not produce tiny results; input values less than or equal to \[1.00000000001100000000B \times 2^{126}\] 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

\[
\text{DEST}[31-0] \leftarrow \text{APPROX}(1.0/(\text{SRC}[31-0]));
\]

* \text{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.
<table>
<thead>
<tr>
<th>Exception Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#SS(0)</td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td>#PF(fault-code)</td>
<td>For a page fault.</td>
</tr>
<tr>
<td>#NM</td>
<td>If TS in CR0 is set.</td>
</tr>
<tr>
<td>#UD</td>
<td>If EM in CR0 is set.</td>
</tr>
<tr>
<td>#AC(0)</td>
<td>If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.</td>
</tr>
<tr>
<td>GP(0)</td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>#NM</td>
<td>If TS in CR0 is set.</td>
</tr>
<tr>
<td>#UD</td>
<td>If EM in CR0 is set.</td>
</tr>
<tr>
<td>#AC(0)</td>
<td>If CPUID feature flag SSE is 0.</td>
</tr>
</tbody>
</table>

**Real-Address Mode Exceptions**

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 32</td>
<td>RDMSR</td>
<td>Load MSR specified by ECX into EDX:EAX.</td>
</tr>
</tbody>
</table>

**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.
- #GP(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.
RDP MC—Read Performance-Monitoring Counters

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 33</td>
<td>RDP MC</td>
<td>Read performance-monitoring counter specified by ECX into EDX:EAX.</td>
</tr>
</tbody>
</table>

**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 RDP MC 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 RDP MC instruction as follows. When the PCE flag is set, the RDP MC 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 RDP MC 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 RDP MC 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 RDP CM instruction.

In the Pentium 4 and Intel Xeon processors, performing back-to-back fast reads are not guaranteed to be monotonic. To guarantee monotonicity on back-to-back reads, a serializing instruction must be placed between the two RDP MC 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 \((ECX=0 \text{ OR } 1) \text{ AND } ((CR4.PCE=1) \text{ OR } (CPL=0) \text{ OR } (CR0.PE=0))\)

THEN

\[
\begin{align*}
\text{EAX} & \leftarrow \text{PMC(ECX)[31:0]}; \\
\text{EDX} & \leftarrow \text{PMC(ECX)[39:32]};
\end{align*}
\]

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 \((ECX[30:0]=0 ... 17) \text{ AND } ((CR4.PCE=1) \text{ OR } (CPL=0) \text{ OR } (CR0.PE=0))\)

THEN IF ECX[31] = 0

THEN

\[
\begin{align*}
\text{EAX} & \leftarrow \text{PMC(ECX[30:0])[31:0]}; \text{ (* 40-bit read *)}; \\
\text{EDX} & \leftarrow \text{PMC(ECX[30:0])[39:32]};
\end{align*}
\]

ELSE IF ECX[31] = 1

THEN

\[
\begin{align*}
\text{EAX} & \leftarrow \text{PMC(ECX[30:0])[31:0]}; \text{ (* 32-bit read *)}; \\
\text{EDX} & \leftarrow 0;
\end{align*}
\]

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

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

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

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.

4-164 Vol. 2B
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>
<thead>
<tr>
<th>Repeat Prefix</th>
<th>Termination Condition 1</th>
<th>Termination Condition 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>REP</td>
<td>ECX=0</td>
<td>None</td>
</tr>
<tr>
<td>REPE/REPZ</td>
<td>ECX=0</td>
<td>ZF=0</td>
</tr>
<tr>
<td>REPNE/REPNZ</td>
<td>ECX=0</td>
<td>ZF=1</td>
</tr>
</tbody>
</table>

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

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;

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 addresses descriptor beyond descriptor 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 their 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 *)
INSTRUCTION SET REFERENCE, N-Z

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;
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);
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(); (* 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.
INSTRUCTION SET REFERENCE, N-Z

#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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F AA</td>
<td>RSM</td>
<td>Resume operation of interrupted program.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 52 lr</td>
<td>RSQRTPS xmm1, xmm2/m128</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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 I* for an illustration of an SIMD single-precision floating-point operation.

The relative error for this approximation is:

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

The RSQRTPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ 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**

$$\text{DEST}[31-0] \leftarrow \text{APPROXIMATE}(1.0/\sqrt{\text{SRC}[31-0]});$$
$$\text{DEST}[63-32] \leftarrow \text{APPROXIMATE}(1.0/\sqrt{\text{SRC}[63-32]});$$
$$\text{DEST}[95-64] \leftarrow \text{APPROXIMATE}(1.0/\sqrt{\text{SRC}[95-64]});$$
$$\text{DEST}[127-96] \leftarrow \text{APPROXIMATE}(1.0/\sqrt{\text{SRC}[127-96]});$$

**Intel C/C++ Compiler Intrinsic Equivalent**

RSQRTPS _m128 _mm_rsqrt_ps(_m128 a)

**SIMD Floating-Point Exceptions**

None.

**Protected Floating-Point 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

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 double-words 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:

\[ |\text{Relative Error}| \leq 1.5 \times 2^{-12} \]

The RSQRTSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an \( \infty \) 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

\[
\text{DEST}[31-0] \leftarrow \text{APPROXIMATE}(1.0/\text{SQRT}(\text{SRC}[31-0]));
\]

* \( \text{DEST}[127-32] \) remains unchanged *

Intel C/C++ Compiler Intrinsic Equivalent

RSQRTSS

\text{__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.
INSTRUCTION SET REFERENCE, N-Z

#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.

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9E</td>
<td>SAHF</td>
<td>Load SF, ZF, AF, PF, and CF from AH into EFLAGS register.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0 /4</td>
<td>SAL r/m8</td>
<td>Multiply r/m8 by 2, 1 time.</td>
</tr>
<tr>
<td>D2 /4</td>
<td>SAL r/m8,CL</td>
<td>Multiply r/m8 by 2, CL times.</td>
</tr>
<tr>
<td>C0 /4</td>
<td>SAL r/m8,imm8</td>
<td>Multiply r/m8 by 2, imm8 times.</td>
</tr>
<tr>
<td>D1 /4</td>
<td>SAL r/m16</td>
<td>Multiply r/m16 by 2, 1 time.</td>
</tr>
<tr>
<td>D3 /4</td>
<td>SAL r/m16,CL</td>
<td>Multiply r/m16 by 2, CL times.</td>
</tr>
<tr>
<td>C1 /4</td>
<td>SAL r/m16,imm8</td>
<td>Multiply r/m16 by 2, imm8 times.</td>
</tr>
<tr>
<td>D1 /4</td>
<td>SAR r/m8</td>
<td>Signed divide* r/m8 by 2, 1 times.</td>
</tr>
<tr>
<td>D2 /7</td>
<td>SAR r/m8,CL</td>
<td>Signed divide* r/m8 by 2, CL times.</td>
</tr>
<tr>
<td>C0 /7</td>
<td>SAR r/m8,imm8</td>
<td>Signed divide* r/m8 by 2, imm8 times.</td>
</tr>
<tr>
<td>D1 /7</td>
<td>SAR r/m16</td>
<td>Signed divide* r/m16 by 2, 1 time.</td>
</tr>
<tr>
<td>D3 /7</td>
<td>SAR r/m16,CL</td>
<td>Signed divide* r/m16 by 2, CL times.</td>
</tr>
<tr>
<td>C1 /7</td>
<td>SAR r/m16,imm8</td>
<td>Signed divide* r/m16 by 2, imm8 times.</td>
</tr>
<tr>
<td>D1 /4</td>
<td>SHL r/m8</td>
<td>Multiply r/m8 by 2, 1 time.</td>
</tr>
<tr>
<td>D2 /4</td>
<td>SHL r/m8,CL</td>
<td>Multiply r/m8 by 2, CL times.</td>
</tr>
<tr>
<td>C0 /4</td>
<td>SHL r/m8,imm8</td>
<td>Multiply r/m8 by 2, imm8 times.</td>
</tr>
<tr>
<td>D1 /4</td>
<td>SHL r/m16</td>
<td>Multiply r/m16 by 2, 1 time.</td>
</tr>
<tr>
<td>D3 /4</td>
<td>SHL r/m16,CL</td>
<td>Multiply r/m16 by 2, CL times.</td>
</tr>
<tr>
<td>C1 /4</td>
<td>SHL r/m16,imm8</td>
<td>Multiply r/m16 by 2, imm8 times.</td>
</tr>
<tr>
<td>D1 /4</td>
<td>SHR r/m8</td>
<td>Unsigned divide r/m8 by 2, 1 time.</td>
</tr>
<tr>
<td>D2 /5</td>
<td>SHR r/m8,CL</td>
<td>Unsigned divide r/m8 by 2, CL times.</td>
</tr>
<tr>
<td>C0 /5</td>
<td>SHR r/m8,imm8</td>
<td>Unsigned divide r/m8 by 2, imm8 times.</td>
</tr>
<tr>
<td>D1 /5</td>
<td>SHR r/m16</td>
<td>Unsigned divide r/m16 by 2, 1 time.</td>
</tr>
<tr>
<td>D3 /5</td>
<td>SHR r/m16,CL</td>
<td>Unsigned divide r/m16 by 2, CL times.</td>
</tr>
<tr>
<td>C1 /5</td>
<td>SHR r/m16,imm8</td>
<td>Unsigned divide r/m16 by 2, imm8 times.</td>
</tr>
<tr>
<td>D1 /5</td>
<td>SHR r/m32</td>
<td>Unsigned divide r/m32 by 2, 1 time.</td>
</tr>
<tr>
<td>D3 /5</td>
<td>SHR r/m32,CL</td>
<td>Unsigned divide r/m32 by 2, CL times.</td>
</tr>
<tr>
<td>C1 /5</td>
<td>SHR r/m32,imm8</td>
<td>Unsigned divide r/m32 by 2, imm8 times.</td>
</tr>
</tbody>
</table>

**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 I).

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 I); 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 I).

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 unde-
defined 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

**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**

\[
\text{DEST} \leftarrow \text{DEST} - (\text{SRC} + \text{CF});
\]
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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AE</td>
<td>SCAS m8</td>
<td>Compare AL with byte at ES:(E)DI and set status flags.</td>
</tr>
<tr>
<td>AF</td>
<td>SCAS m16</td>
<td>Compare AX with word at ES:(E)DI and set status flags.</td>
</tr>
<tr>
<td>AF</td>
<td>SCAS m32</td>
<td>Compare EAX with doubleword at ES(E)DI and set status flags.</td>
</tr>
<tr>
<td>AE</td>
<td>SCASB</td>
<td>Compare AL with byte at ES:(E)DI and set status flags.</td>
</tr>
<tr>
<td>AF</td>
<td>SCASW</td>
<td>Compare AX with word at ES:(E)DI and set status flags.</td>
</tr>
<tr>
<td>AF</td>
<td>SCASD</td>
<td>Compare EAX with doubleword at ES:(E)DI and set status flags.</td>
</tr>
</tbody>
</table>

**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 comparison)
  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.
SETcc—Set Byte on Condition

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

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 SETcc 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 SETcc 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

**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 to 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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 /0</td>
<td>SGDT m</td>
<td>Store GDTR to m.</td>
</tr>
</tbody>
</table>

**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 OperandsSize = 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.
INSTRUCTION SET REFERENCE, N-Z

SHL/SHR—Shift Instructions
See entry for SAL/SAR/SHL/SHR—Shift.
SHLD—Double Precision Shift Left

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F A4</td>
<td>SHLD r/m16, r16, imm8</td>
<td>Shift r/m16 to left imm8 places while shifting bits from r16 in from the right.</td>
</tr>
<tr>
<td>0F A5</td>
<td>SHLD r/m16, r16, CL</td>
<td>Shift r/m16 to left CL places while shifting bits from r16 in from the right.</td>
</tr>
<tr>
<td>0F A4</td>
<td>SHLD r/m32, r32, imm8</td>
<td>Shift r/m32 to left imm8 places while shifting bits from r32 in from the right.</td>
</tr>
<tr>
<td>0F A5</td>
<td>SHLD r/m32, r32, CL</td>
<td>Shift r/m32 to left CL places while shifting bits from r32 in from the right.</td>
</tr>
</tbody>
</table>

**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;

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

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

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.
**INSTRUCTION SET REFERENCE, N-Z**

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F C6 /r ib</td>
<td>SHUFPD xmm1, xmm2/m128, imm8</td>
<td>Shuffle packed double-precision floating-point values selected by imm8 from xmm1 and xmm1/m128 to xmm1.</td>
</tr>
</tbody>
</table>

**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](image)

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]; Fl;
IF SELECT[1] = 0
    THEN DEST[127-64] ← SRC[63-0];
    ELSE DEST[127-64] ← SRC[127-64]; Fl;
Intel C/C++ Compiler Intrinsic Equivalent

SHUFDPD  __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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F C6 /r ib</td>
<td>SHUFPS xmm1, xmm2/m128, imm8</td>
<td>Shuffle packed single-precision floating-point values selected by imm8 from xmm1 and xmm1/m128 to xmm1.</td>
</tr>
</tbody>
</table>

**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 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](image)

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

**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.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 /1</td>
<td>SIDT m</td>
<td>Store IDTR to m.</td>
</tr>
</tbody>
</table>
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

**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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 /4</td>
<td>SMSW r/m16</td>
<td>Store machine status word to r/m16.</td>
</tr>
<tr>
<td>0F 01 /4</td>
<td>SMSW r32/m16</td>
<td>Store machine status word in low-order 16 bits of r32/m16; high-order 16 bits of r32 are undefined.</td>
</tr>
</tbody>
</table>

**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 Intel86 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

**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.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 51 /r</td>
<td>SQRTPD xmm1, xmm2/m128</td>
<td>Compute the square root of the packed double-precision floating-point values in xmm2/m128 and store the results in xmm1.</td>
</tr>
</tbody>
</table>
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

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

\[
\begin{align*}
\text{DEST}[31-0] & \leftarrow \text{SQRT}(\text{SRC}[31-0]); \\
\text{DEST}[63-32] & \leftarrow \text{SQRT}(\text{SRC}[63-32]); \\
\text{DEST}[95-64] & \leftarrow \text{SQRT}(\text{SRC}[95-64]); \\
\text{DEST}[127-96] & \leftarrow \text{SQRT}(\text{SRC}[127-96]);
\end{align*}
\]

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.
SQR TSD—Compute Square Root of Scalar Double-Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 51 /r</td>
<td>SQR TSD xmm1, xmm2/m64</td>
<td>Compute the square root of the low double-precision floating-point value in xmm2/m64 and store the results in xmm1.</td>
</tr>
</tbody>
</table>

**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 I* for an illustration of a scalar double-precision floating-point operation.

**Operation**

DEST[63-0] ← SQR T(SRC[63-0]);
* DEST[127-64] remains unchanged *;

**Intel C/C++ Compiler Intrinsic Equivalent**

SQR TSD __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 OSXM MEXCPT in CR4 is 1.
- #UD If an unmasked SIMD floating-point exception and OSXM MEXCPT 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.
SQRTPSS—Compute Square Root of Scalar Single-Precision Floating-Point Value

**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**
\[
\text{DEST}[31-0] \leftarrow \sqrt{\text{SRC}[31-0]};
\]
*DEST[127-64] remains unchanged*;

**Intel C/C++ Compiler Intrinsic Equivalent**

```
SQRTPSS
_mm128 _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

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FD</td>
<td>STD</td>
<td>Set DF flag.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FB</td>
<td>STI</td>
<td>Set interrupt flag; external, maskable interrupts enabled at the end of the next instruction.</td>
</tr>
</tbody>
</table>

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. 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-delays 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.
### 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 *)

---

### Table 4-2. Decision Table for STI Results

<table>
<thead>
<tr>
<th>PE</th>
<th>VM</th>
<th>IOPL</th>
<th>CPL</th>
<th>PVI</th>
<th>VIP</th>
<th>VME</th>
<th>STI Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>IF = 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>≥ CPL</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>IF = 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>&lt; CPL</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>VIF = 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>&lt; CPL</td>
<td>&lt; 3</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>GP Fault</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>&lt; CPL</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>GP Fault</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>&lt; CPL</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>GP Fault</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>IF = 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>&lt; 3</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>VIF = 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>&lt; 3</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>GP Fault</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>&lt; 3</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>GP Fault</td>
</tr>
</tbody>
</table>

X = This setting has no impact.
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

 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.

Description

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.
INSTRUCTION SET REFERENCE, N-Z

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>STOS m8</td>
<td>Store AL at address ES:(E)DI.</td>
</tr>
<tr>
<td>AB</td>
<td>STOS m16</td>
<td>Store AX at address ES:(E)DI.</td>
</tr>
<tr>
<td>AB</td>
<td>STOS m32</td>
<td>Store EAX at address ES:(E)DI.</td>
</tr>
<tr>
<td>AA</td>
<td>STOSB</td>
<td>Store AL at address ES:(E)DI.</td>
</tr>
<tr>
<td>AB</td>
<td>STOSW</td>
<td>Store AX at address ES:(E)DI.</td>
</tr>
<tr>
<td>AB</td>
<td>STOSD</td>
<td>Store EAX at address ES:(E)DI.</td>
</tr>
</tbody>
</table>

**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

\[
\text{DEST} \leftarrow \text{AL}; \\
\text{THEN IF DF} = 0 \\
\text{THEN (E)DI} \leftarrow (\text{E})\text{DI} + 1; \quad \text{ELSE (E)DI} \leftarrow (\text{E})\text{DI} - 1; \quad \text{FI}; \\
\text{ELSE IF (word store)} \quad \\
\text{DEST} \leftarrow \text{AX}; \\
\text{THEN IF DF} = 0 \\
\text{THEN (E)DI} \leftarrow (\text{E})\text{DI} + 2; \quad \text{ELSE (E)DI} \leftarrow (\text{E})\text{DI} - 2; \quad \text{FI}; \\
\text{ELSE (* doubleword store *) } \\
\text{DEST} \leftarrow \text{EAX}; \\
\text{THEN IF DF} = 0 \\
\text{THEN (E)DI} \leftarrow (\text{E})\text{DI} + 4; \quad \text{ELSE (E)DI} \leftarrow (\text{E})\text{DI} - 4; \quad \text{FI}; \\
\text{FI}; \\
\text{FI};
\]

**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

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

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

\[
\text{DEST} \leftarrow \text{DEST} - \text{SRC};
\]

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.
INSTRUCTION SET REFERENCE, N-Z

SUBPD—Subtract Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 5C</td>
<td>SUBPD xmm1, xmm2/m128</td>
<td>Subtract packed double-precision floating-point values in xmm2/m128 from xmm1.</td>
</tr>
</tbody>
</table>

**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

**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**

\[
\begin{align*}
\text{DEST}[31-0] &\leftarrow \text{DEST}[31-0] - \text{SRC}[31-0]; \\
\text{DEST}[63-32] &\leftarrow \text{DEST}[63-32] - \text{SRC}[63-32]; \\
\text{DEST}[95-64] &\leftarrow \text{DEST}[95-64] - \text{SRC}[95-64]; \\
\text{DEST}[127-96] &\leftarrow \text{DEST}[127-96] - \text{SRC}[127-96];
\end{align*}
\]

**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.
Real-Address Mode Exceptions

#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.

#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

### 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
\[
\text{DEST}[63-0] \leftarrow \text{DEST}[63-0] - \text{SRC}[63-0];
\]
\* DEST[127-64] remains unchanged \*

### Intel C/C++ Compiler Intrinsic Equivalent
\[
\text{SUBSD} \quad \text{__m128d } \text{_mm_sub_sd} (\text{m128d } a, \text{m128d } b)
\]

### 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 OSXMEXCPT in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMEXCPT 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

**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**

\[
\text{DEST}[31-0] \leftarrow \text{DEST}[31-0] - \text{SRC}[31-0]
\]

* \text{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 OSXMMEXCPTE in CR4 is 1.

#UD If an unmasked SIMD floating-point exception and OSXMMEXCPTE 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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 34</td>
<td>SYSENTER</td>
<td>Fast call to privilege level 0 system procedures.</td>
</tr>
</tbody>
</table>

**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>
<thead>
<tr>
<th>MSR</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSENTER_CS_MSR</td>
<td>174H</td>
</tr>
<tr>
<td>SYSENTER_ESP_MSR</td>
<td>175H</td>
</tr>
<tr>
<td>SYSENTER_EIP_MSR</td>
<td>176H</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 35</td>
<td>SYSEXIT</td>
<td>Fast return to privilege level 3 user code.</td>
</tr>
</tbody>
</table>

**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

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
\[
\text{TEMP} \leftarrow \text{SRC1 AND SRC2}; \\
\text{SF} \leftarrow \text{MSB(TEMP)}; \\
\text{IF TEMP} = 0 \\
\text{THEN ZF} \leftarrow 1; \\
\text{ELSE ZF} \leftarrow 0; \\
\text{FI}; \\
\text{PF} \leftarrow \text{BitwiseXNOR(TEMP[0:7])}; \\
\text{CF} \leftarrow 0; \\
\text{OF} \leftarrow 0; \\
(*\text{AF is Undefined}*)
\]

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

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#GP(0)</td>
<td>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.</td>
</tr>
<tr>
<td>#SS(0)</td>
<td>If a memory operand effective address is outside the SS segment limit.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A8 ib</td>
<td>TEST AL, imm8</td>
<td>AND imm8 with AL; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>A9 iw</td>
<td>TEST AX, imm16</td>
<td>AND imm16 with AX; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>A9 id</td>
<td>TEST EAX, imm32</td>
<td>AND imm32 with EAX; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>F6 /0 ib</td>
<td>TEST r/m8, imm8</td>
<td>AND imm8 with r/m8; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>F7 /0 iw</td>
<td>TEST r/m16, imm16</td>
<td>AND imm16 with r/m16; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>F7 /0 id</td>
<td>TEST r/m32, imm32</td>
<td>AND imm32 with r/m32; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>84 /r</td>
<td>TEST r/m8, r8</td>
<td>AND r8 with r/m8; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>85 /r</td>
<td>TEST r/m16, r16</td>
<td>AND r16 with r/m16; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>85 /r</td>
<td>TEST r/m32, r32</td>
<td>AND r32 with r/m32; set SF, ZF, PF according to result.</td>
</tr>
</tbody>
</table>
#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

**Description**

Performs and unordered compare of the double-precision floating-point values in the low quad-words 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)
```

**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.
```
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

**Description**
Performs and unordered compare of the single-precision floating-point values in the low double-words 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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 0B</td>
<td>UD2</td>
<td>Raise invalid opcode exception.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 15 /r</td>
<td>UNPCKHPD xmm1, xmm2/m128</td>
<td>Unpack and interleave double-precision floating-point values from the high quadwords of xmm1 and xmm2/m128.</td>
</tr>
</tbody>
</table>

**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.

![Diagram](image)

**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

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.

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

### 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](image)

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

Perform 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.

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 00 /4</td>
<td>VERR r/m16</td>
<td>Set ZF=1 if segment specified with r/m16 can be read.</td>
</tr>
<tr>
<td>0F 00 /5</td>
<td>VERW r/m16</td>
<td>Set ZF=1 if segment specified with r/m16 can be written.</td>
</tr>
</tbody>
</table>

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(DescriptorType) ≠ 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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9B</td>
<td>WAIT</td>
<td>Check pending unmasked floating-point exceptions.</td>
</tr>
<tr>
<td>9B</td>
<td>FWAIT</td>
<td>Check pending unmasked floating-point exceptions.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 09</td>
<td>WBINVD</td>
<td>Write back and flush internal caches; initiate writing-back and flushing of external caches.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 30</td>
<td>WRMSR</td>
<td>Write the value in EDX:EAX to MSR specified by ECX</td>
</tr>
</tbody>
</table>

**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.
INSTRUCTION SET REFERENCE, N-Z

XADD—Exchange and Add

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F C0</td>
<td>XADD r/m8, r8</td>
<td>Exchange r8 and r/m8; load sum into r/m8.</td>
</tr>
<tr>
<td>0F C1</td>
<td>XADD r/m16, r16</td>
<td>Exchange r16 and r/m16; load sum into r/m16.</td>
</tr>
<tr>
<td>0F C1</td>
<td>XADD r/m32, r32</td>
<td>Exchange r32 and r/m32; load sum into r/m32.</td>
</tr>
</tbody>
</table>

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.
INSTRUCTION SET REFERENCE, N-Z

XCHG—Exchange Register/Memory with Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>90+rw</td>
<td>XCHG AX, 16</td>
<td>Exchange r16 with AX.</td>
</tr>
<tr>
<td>90+nw</td>
<td>XCHG r16, X</td>
<td>Exchange AX with r16.</td>
</tr>
<tr>
<td>90+rd</td>
<td>XCHG EAX, r32</td>
<td>Exchange r32 with EAX.</td>
</tr>
<tr>
<td>90+rd</td>
<td>XCHG r32, EAX</td>
<td>Exchange EAX with r32.</td>
</tr>
<tr>
<td>86 lr</td>
<td>XCHG r/m8, r8</td>
<td>Exchange r8 (byte register) with byte from r/m8.</td>
</tr>
<tr>
<td>86 lr</td>
<td>XCHG r8, r/m8</td>
<td>Exchange byte from r/m8 with r8 (byte register).</td>
</tr>
<tr>
<td>87 lr</td>
<td>XCHG r/m16, r16</td>
<td>Exchange r16 with word from r/m16.</td>
</tr>
<tr>
<td>87 lr</td>
<td>XCHG r16, r/m16</td>
<td>Exchange word from r/m16 with r16.</td>
</tr>
<tr>
<td>87 lr</td>
<td>XCHG r/m32, r32</td>
<td>Exchange r32 with doubleword from r/m32.</td>
</tr>
<tr>
<td>87 lr</td>
<td>XCHG r32, r/m32</td>
<td>Exchange doubleword from r/m32 with r32.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D7</td>
<td>XLAT m8</td>
<td>Set AL to memory byte DS:((E)BX + unsigned AL).</td>
</tr>
<tr>
<td>D7</td>
<td>XLATB</td>
<td>Set AL to memory byte DS:((E)BX + unsigned AL).</td>
</tr>
</tbody>
</table>

**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

**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.
<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#SS(0)</td>
<td>If a memory operand effective address is outside the SS segment limit.</td>
</tr>
<tr>
<td>#PF(fault-code)</td>
<td>If a page fault occurs.</td>
</tr>
<tr>
<td>#AC(0)</td>
<td>If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>

**Real-Address Mode Exceptions**

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#GP</td>
<td>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</td>
</tr>
<tr>
<td>#SS</td>
<td>If a memory operand effective address is outside the SS segment limit.</td>
</tr>
</tbody>
</table>

**Virtual-8086 Mode Exceptions**

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#GP(0)</td>
<td>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</td>
</tr>
<tr>
<td>#SS(0)</td>
<td>If a memory operand effective address is outside the SS segment limit.</td>
</tr>
<tr>
<td>#PF(fault-code)</td>
<td>If a page fault occurs.</td>
</tr>
<tr>
<td>#AC(0)</td>
<td>If alignment checking is enabled and an unaligned memory reference is made.</td>
</tr>
</tbody>
</table>
XORPD—Bitwise Logical XOR for Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 57 /r</td>
<td>XORPD xmm1, xmm2/m128</td>
<td>Bitwise exclusive-OR of xmm2/m128 and xmm1.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 57 /r</td>
<td>XORPS xmm1, xmm2/m128</td>
<td>Bitwise exclusive-OR of xmm2/m128 and xmm1.</td>
</tr>
</tbody>
</table>

**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.
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, MOV, CMP, OUTS, or LODS).

Y Memory addressed by the ES:DI register pair (for example, MOV, CMP, 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

<table>
<thead>
<tr>
<th>LSB address</th>
<th>03</th>
<th>05</th>
<th>00</th>
<th>00</th>
<th>MSB address</th>
<th>00</th>
</tr>
</thead>
</table>

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>
<thead>
<tr>
<th>Symbol</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>1A</td>
<td>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”).</td>
</tr>
<tr>
<td>1B</td>
<td>Use the 0F0B opcode (UD2 instruction) or the 0FB9H opcode when deliberately trying to generate an invalid opcode exception (#UD).</td>
</tr>
<tr>
<td>1C</td>
<td>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.)</td>
</tr>
<tr>
<td>1D</td>
<td>The instruction represented by this opcode expression does not have a ModR/M byte following the primary opcode.</td>
</tr>
<tr>
<td>1E</td>
<td>Valid encoding for the r/m field of the ModR/M byte is shown in parenthesis.</td>
</tr>
<tr>
<td>1F</td>
<td>The instruction represented by this opcode expression does not support both source and destination operands to be registers.</td>
</tr>
<tr>
<td>1G</td>
<td>When the source operand is a register, it must be an XMM register.</td>
</tr>
<tr>
<td>1H</td>
<td>The instruction represented by this opcode expression does not support any operand to be a memory location.</td>
</tr>
<tr>
<td>1J</td>
<td>The instruction represented by this opcode expression does not support register operand.</td>
</tr>
<tr>
<td>1K</td>
<td>Valid encoding for the reg/opcode field of the ModR/M byte is shown in parenthesis.</td>
</tr>
</tbody>
</table>
Table A-2. One-byte Opcode Map† ††

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADD</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL, Ib&lt;sup&gt;10&lt;/sup&gt;</td>
<td>eAX, Iv&lt;sup&gt;10&lt;/sup&gt;</td>
</tr>
<tr>
<td>1</td>
<td>ADC</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL, Ib&lt;sup&gt;10&lt;/sup&gt;</td>
<td>eAX, Iv&lt;sup&gt;10&lt;/sup&gt;</td>
</tr>
<tr>
<td>2</td>
<td>AND</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL, Ib&lt;sup&gt;10&lt;/sup&gt;</td>
<td>eAX, Iv&lt;sup&gt;10&lt;/sup&gt;</td>
</tr>
<tr>
<td>3</td>
<td>XOR</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL, Ib&lt;sup&gt;10&lt;/sup&gt;</td>
<td>eAX, Iv&lt;sup&gt;10&lt;/sup&gt;</td>
</tr>
<tr>
<td>4</td>
<td>INC</td>
<td>eAX&lt;sup&gt;10&lt;/sup&gt;</td>
<td>eCX&lt;sup&gt;10&lt;/sup&gt;</td>
<td>eDX&lt;sup&gt;10&lt;/sup&gt;</td>
<td>eBX&lt;sup&gt;10&lt;/sup&gt;</td>
<td>eSP&lt;sup&gt;10&lt;/sup&gt;</td>
<td>eBP&lt;sup&gt;10&lt;/sup&gt;</td>
</tr>
<tr>
<td>5</td>
<td>PUSH</td>
<td>eAX</td>
<td>eCX</td>
<td>eDX</td>
<td>eBX</td>
<td>eSP</td>
<td>eBP</td>
</tr>
<tr>
<td>6</td>
<td>POP</td>
<td>PUSHA/ PUSHAD&lt;sup&gt;10&lt;/sup&gt;</td>
<td>POPA/ POPAD&lt;sup&gt;10&lt;/sup&gt;</td>
<td>BOUND</td>
<td>Gv, Ma</td>
<td>ARPL</td>
<td>Ew, Gw</td>
</tr>
<tr>
<td>7</td>
<td>Jcc, Jb - Short-displacement jump on condition</td>
<td>O&lt;sup&gt;10&lt;/sup&gt;</td>
<td>NO&lt;sup&gt;10&lt;/sup&gt;</td>
<td>B/NAE/NC&lt;sup&gt;10&lt;/sup&gt;</td>
<td>NB/AE/NC&lt;sup&gt;10&lt;/sup&gt;</td>
<td>Z/E&lt;sup&gt;10&lt;/sup&gt;</td>
<td>NZ/NE&lt;sup&gt;10&lt;/sup&gt;</td>
</tr>
<tr>
<td>8</td>
<td>Immediate Grp</td>
<td>1&lt;sup&gt;1A&lt;/sup&gt;</td>
<td>TEST</td>
<td>XCHG</td>
<td>Eb, Ib</td>
<td>Ev, Iv</td>
<td>Eb, Ib</td>
</tr>
<tr>
<td>9</td>
<td>NOP</td>
<td>XCHG word or double-word register with eAX&lt;sup&gt;10&lt;/sup&gt;</td>
<td>eAX</td>
<td>eCX</td>
<td>eDX</td>
<td>eBX</td>
<td>eSP</td>
</tr>
<tr>
<td>A</td>
<td>MOV</td>
<td>AL, Ob</td>
<td>eAX, Ov</td>
<td>Ob, AL</td>
<td>Ov, eAX</td>
<td>MOV/S MOVSB</td>
<td>MOV/S MOVSW</td>
</tr>
<tr>
<td>B</td>
<td>MOVS/MOVS/MOVS/</td>
<td>MOV immediate byte into byte register&lt;sup&gt;10&lt;/sup&gt;</td>
<td>AL</td>
<td>CL</td>
<td>DL</td>
<td>BL</td>
<td>AH</td>
</tr>
<tr>
<td>C</td>
<td>Shift Grp</td>
<td>2&lt;sup&gt;1A&lt;/sup&gt;</td>
<td>Eb, Ib</td>
<td>Ev, Ib</td>
<td>RET&lt;sup&gt;10&lt;/sup&gt;</td>
<td>RET&lt;sup&gt;10&lt;/sup&gt;</td>
<td>LES</td>
</tr>
<tr>
<td>D</td>
<td>Shift Grp</td>
<td>2&lt;sup&gt;1A&lt;/sup&gt;</td>
<td>Eb, 1</td>
<td>Ev, 1</td>
<td>Eb, CL</td>
<td>Ev, CL</td>
<td>AAM</td>
</tr>
<tr>
<td>E</td>
<td>LOOPNE// LOOPNZ</td>
<td>Jb&lt;sup&gt;10&lt;/sup&gt;</td>
<td>LOOP// LOOPZ</td>
<td>Jb&lt;sup&gt;10&lt;/sup&gt;</td>
<td>LOOP</td>
<td>Jb&lt;sup&gt;10&lt;/sup&gt;</td>
<td>JECXZ</td>
</tr>
<tr>
<td>F</td>
<td>LOCK</td>
<td>REPNE/</td>
<td>REP/REPE</td>
<td>Prefix</td>
<td>REP/</td>
<td>REP/</td>
<td>Prefix</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>OR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>89AB</td>
<td>C</td>
<td>D</td>
<td>EF</td>
<td></td>
<td></td>
<td>PUSH</td>
<td>Escape opcode to 2-byte</td>
</tr>
<tr>
<td>0O</td>
<td>R</td>
<td>P</td>
<td>PUSH</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>U</td>
<td>S</td>
<td>PUSH</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td>D</td>
<td>PUSH</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>SBB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0O</td>
<td>R</td>
<td>P</td>
<td>U</td>
<td>S</td>
<td>SBB</td>
<td>PUSH</td>
<td>Escape opcode to 2-byte</td>
</tr>
<tr>
<td>DS</td>
<td>P</td>
<td>U</td>
<td>S</td>
<td>SBB</td>
<td>PUSH</td>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>U</td>
<td>S</td>
<td>SBB</td>
<td>DS</td>
<td>PUSH</td>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>SUB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0O</td>
<td>R</td>
<td>P</td>
<td>U</td>
<td>SUB</td>
<td>SEG-C</td>
<td>SEG-C</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td>D</td>
<td>PUSH</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>CMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0O</td>
<td>R</td>
<td>P</td>
<td>U</td>
<td>CMP</td>
<td>SEG-D</td>
<td>SEG-D</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td>D</td>
<td>PUSH</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>DEC general register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0O</td>
<td>R</td>
<td>P</td>
<td>DEC general register</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td>D</td>
<td>DEC general register</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>POP into general register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0O</td>
<td>R</td>
<td>P</td>
<td>POP into general register</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td>D</td>
<td>POP into general register</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Jcc, Jb - Short displacement jump on condition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0O</td>
<td>R</td>
<td>P</td>
<td>Jcc, Jb - Short displacement jump on condition</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td>D</td>
<td>Jcc, Jb - Short displacement jump on condition</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
</tr>
<tr>
<td>0O</td>
<td>R</td>
<td>P</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td>D</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>CBW/</td>
<td>CBW/</td>
<td>CBW/</td>
<td>CBW/</td>
<td>CBW/</td>
<td>CBW/</td>
</tr>
<tr>
<td>0O</td>
<td>R</td>
<td>P</td>
<td>CBW/</td>
<td>CBW/</td>
<td>CBW/</td>
<td>CBW/</td>
<td>CBW/</td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td>D</td>
<td>CBW/</td>
<td>CBW/</td>
<td>CBW/</td>
<td>CBW/</td>
<td>CBW/</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>TEST</td>
<td>TEST</td>
<td>TEST</td>
<td>TEST</td>
<td>TEST</td>
<td>TEST</td>
</tr>
<tr>
<td>0O</td>
<td>R</td>
<td>P</td>
<td>TEST</td>
<td>TEST</td>
<td>TEST</td>
<td>TEST</td>
<td>TEST</td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td>D</td>
<td>TEST</td>
<td>TEST</td>
<td>TEST</td>
<td>TEST</td>
<td>TEST</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>MOV immediate word or double into word or double register</td>
<td>MOV immediate word or double into word or double register</td>
<td>MOV immediate word or double into word or double register</td>
<td>MOV immediate word or double into word or double register</td>
<td>MOV immediate word or double into word or double register</td>
<td>MOV immediate word or double into word or double register</td>
</tr>
<tr>
<td>0O</td>
<td>R</td>
<td>P</td>
<td>MOV immediate word or double into word or double register</td>
<td>MOV immediate word or double into word or double register</td>
<td>MOV immediate word or double into word or double register</td>
<td>MOV immediate word or double into word or double register</td>
<td>MOV immediate word or double into word or double register</td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td>D</td>
<td>MOV immediate word or double into word or double register</td>
<td>MOV immediate word or double into word or double register</td>
<td>MOV immediate word or double into word or double register</td>
<td>MOV immediate word or double into word or double register</td>
<td>MOV immediate word or double into word or double register</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>ENTER</td>
<td>ENTER</td>
<td>ENTER</td>
<td>ENTER</td>
<td>ENTER</td>
<td>ENTER</td>
</tr>
<tr>
<td>0O</td>
<td>R</td>
<td>P</td>
<td>ENTER</td>
<td>ENTER</td>
<td>ENTER</td>
<td>ENTER</td>
<td>ENTER</td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td>D</td>
<td>ENTER</td>
<td>ENTER</td>
<td>ENTER</td>
<td>ENTER</td>
<td>ENTER</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>ESC (Escape to coprocessor instruction set)</td>
<td>ESC (Escape to coprocessor instruction set)</td>
<td>ESC (Escape to coprocessor instruction set)</td>
<td>ESC (Escape to coprocessor instruction set)</td>
<td>ESC (Escape to coprocessor instruction set)</td>
<td>ESC (Escape to coprocessor instruction set)</td>
</tr>
<tr>
<td>0O</td>
<td>R</td>
<td>P</td>
<td>ESC (Escape to coprocessor instruction set)</td>
<td>ESC (Escape to coprocessor instruction set)</td>
<td>ESC (Escape to coprocessor instruction set)</td>
<td>ESC (Escape to coprocessor instruction set)</td>
<td>ESC (Escape to coprocessor instruction set)</td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td>D</td>
<td>ESC (Escape to coprocessor instruction set)</td>
<td>ESC (Escape to coprocessor instruction set)</td>
<td>ESC (Escape to coprocessor instruction set)</td>
<td>ESC (Escape to coprocessor instruction set)</td>
<td>ESC (Escape to coprocessor instruction set)</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>CALL</td>
<td>CALL</td>
<td>CALL</td>
<td>CALL</td>
<td>CALL</td>
<td>CALL</td>
</tr>
<tr>
<td>0O</td>
<td>R</td>
<td>P</td>
<td>CALL</td>
<td>CALL</td>
<td>CALL</td>
<td>CALL</td>
<td>CALL</td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td>D</td>
<td>CALL</td>
<td>CALL</td>
<td>CALL</td>
<td>CALL</td>
<td>CALL</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>CLC</td>
<td>CLC</td>
<td>CLC</td>
<td>CLC</td>
<td>CLC</td>
<td>CLC</td>
</tr>
<tr>
<td>0O</td>
<td>R</td>
<td>P</td>
<td>CLC</td>
<td>CLC</td>
<td>CLC</td>
<td>CLC</td>
<td>CLC</td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td>D</td>
<td>CLC</td>
<td>CLC</td>
<td>CLC</td>
<td>CLC</td>
<td>CLC</td>
</tr>
</tbody>
</table>
Table A-3. Two-byte Opcode Map (First Byte is 0FH)†††

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Grp 6Å</td>
<td>Grp 7Å</td>
<td>LAR Gv, Ew</td>
<td>LSL Gv, Ew</td>
<td></td>
<td>CLTS12</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>MOVUPS Vps, Vps MOVSS (F3) Vss, Vss MOVUPD (66) Vpd, Vpd MOVSD (F2) Vsd, Vsd</td>
<td>MOVUPS Vps, Vps MOVSS (F3) Vss, Vss MOVUPD (66) Vpd, Vpd MOVSD (F2) Vsd, Vsd</td>
<td>MOVLPS Vq, Ms1F MOVLDPD (66) Vq, Ms1F MOVHLPSP Vps, Vps MOVDDUP (F2) Vq, Wq1G MOVSBDUP (F3) Vps, Vps</td>
<td>MOVLPS Vq, Ms1F MOVLDPD (66) Vq, Ms1F MOVHLPSP Vps, Vps MOVDDUP (F2) Vq, Wq1G MOVSBDUP (F3) Vps, Vps</td>
<td>UNPKLPS Vps, Vps UNPKLPD (66) Vpd, Vpd</td>
<td>UNPKHPS Vps, Vps UNPKHPD (66) Vpd, Vpd</td>
<td>MOVHPS Vq, Ms1F MOVHDPD (66) Vq, Ms1F MOVHLPSP Vps, Vps MOVSHDUP (F3) Vps, Vps</td>
</tr>
<tr>
<td>2</td>
<td>MOV Rd, Cd1H</td>
<td>MOV Rd, Dd1H</td>
<td>MOV Cd, Rd1H</td>
<td>MOV Cd, Rd1H</td>
<td>MOV Cd, Rd1H</td>
<td>MOV Rd, Td†††</td>
<td>MOV Td, Rd†††</td>
</tr>
<tr>
<td>3</td>
<td>WRMGR10</td>
<td>RDTSC10</td>
<td>RDTSC10</td>
<td>SYSENTER10</td>
<td>SYSEXIT10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>CMOVcc, (Gv, Ev) - Conditional Move</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>MOVMSKPS Gd, Vps1H MOVMSKPD (66) Gd, Vpd1H</td>
<td>SORTPS Vps, Vps SORTSS (F3) Vss, Vss SORTPD (66) Vpd, Vpd SORTSS (F2) Vsd, Vsd</td>
<td>RSORTPS Vps, Vps RSORTSS (F3) Vss, Vss</td>
<td>RCPS Vps, Vps RCPS (F3) Vss, Vss</td>
<td>ANDPS Vps, Vps ANDPD (66) Vpd, Vpd</td>
<td>ANDNPS Vps, Vps ANDNPD (66) Vpd, Vpd</td>
<td>ORPS Vps, Vps ORPD (66) Vpd, Vpd</td>
</tr>
<tr>
<td>6</td>
<td>PUNPCKLBW Pq, Qd PUNPCKLBW (66) Vdq, Wdq</td>
<td>PUNPCKLWD Pq, Qd PUNPCKLWD (66) Vdq, Wdq</td>
<td>PUNPCKLDQ Pq, Qd PUNPCKLDQ (66) Vdq, Wdq</td>
<td>PACKSWB Pq, Qq PACKSWB (66) Vdq, Wdq</td>
<td>PCMPTGB Pq, Qq PCMPTGB (66) Vdq, Wdq</td>
<td>PCMPTGW Pq, Qq PCMPTGW (66) Vdq, Wdq</td>
<td>PACKUSWB Pq, Qq PACKUSWB (66) Vdq, Wdq</td>
</tr>
<tr>
<td>7</td>
<td>PSHUFW Pq, Qq, Ib PSHUFW (66) Vdq, Wdq, Ib PSHUFW (F3) Vdq, Wdq, Ib PSHUFHW (F2) Vdq, Wdq, Ib (Grp 121Å)</td>
<td>(Grp 141Å)</td>
<td>PCMPIEQB Pq, Qq PCMPIEQB (66) Vdq, Wdq</td>
<td>PCMPIEQW Pq, Qq PCMPIEQW (66) Vdq, Wdq</td>
<td>PCMPIEQD Pq, Qq PCMPIEQD (66) Vdq, Wdq</td>
<td>PCMPIEQD Pq, Qq PCMPIEQD (66) Vdq, Wdq</td>
<td>EMMS10</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>INVD</td>
<td>WDINVD</td>
<td>UD2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>PREFETCH</td>
<td>Gp 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>MOVAPS Vps, Wps</td>
<td>MOVAPS Vps, Wps</td>
<td>CVTSS2SS (F3)</td>
<td>Vps, Ed CVTTP2PD (66) Vpd, Qq CVTS2SD (F2) Vsd, Ed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOVAPS Vps, Wps</td>
<td>MOVAPS Wps, Vps MOVAPD (66) Vpd, Wpd</td>
<td>MOVVTPS Mps, Vps MOVNTPD (66) Mp, Vpd 1F</td>
<td>CVTSS2SI (F3) Gd, Wsd CVTP2D2PI (66) Pq, Wpd CVTS2DI (F2) Gd, Wsd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOVAPS Wps, Vps MOVAPD (66) Wpd, Vpd</td>
<td>MOVAPS Vps, Qq CVTSS2SS (F3) Vss, Ed CVTTP2PD (66) Vpd, Qq CVTS2SD (F2) Vsd, Ed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOVAPS Vps, Wps</td>
<td>MOVAPS Wps, Vps CVTSS2SI (F3) Vps, Ed CVTTS2DI (F2) Vpd, Wpd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOVAPS Wps, Vps MOVAPD (66) Wpd, Vpd</td>
<td>MOVAPS Wps, Vps CVTSS2SI (F3) Vps, Ed CVTTS2DI (F2) Vpd, Wpd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOVAPS Vps, Wps MOVAPD (66) Vpd, Wpd</td>
<td>CVTSS2SI (F3) Vps, Ed CVTTS2DI (F2) Vpd, Wpd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOVAPS Wps, Vps MOVAPD (66) Wpd, Vpd</td>
<td>CVTSS2SI (F3) Vps, Ed CVTTS2DI (F2) Vpd, Wpd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>CMOVcc(Gv, Ev) - Conditional Move</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ADDPS Vps, Wps ADDSS (F3) Vss, Wss ADDPD (66) Vpd, Wpd ADDSD (F2) Vsd, Wsd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MULPS Vps, Wps MULLSS (F3) Vss, Wss MULPD (66) Vpd, Wpd MULSD (F2) Vsd, Wsd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVTSS2PD Vps, Qq CVTSS2DI (F3) Vss, Ed CVTTP2PD (66) Vpd, Qq CVTS2SD (F2) Vsd, Ed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVTSS2PD Vps, Qq CVTSS2DI (F3) Vss, Ed CVTTP2PD (66) Vpd, Qq CVTS2SD (F2) Vsd, Ed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVTSS2PD Vps, Qq CVTSS2DI (F3) Vss, Ed CVTTP2PD (66) Vpd, Qq CVTS2SD (F2) Vsd, Ed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVTSS2PD Vps, Qq CVTSS2DI (F3) Vss, Ed CVTTP2PD (66) Vpd, Qq CVTS2SD (F2) Vsd, Ed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>PUNPCKHBW Pq, Qq</td>
<td>PUNPCKHBW Q (66) Vdq, Wdq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PUNPCKHWD Pq, Qq PUNPCKHWD (66) Vdq, Wdq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PUNPCKHDQ Pq, Qq PUNPCKHDQ (66) Vdq, Wdq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PACKSSDW Pq, Qq PACKSSDW (66) Vdq, Wdq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PUNPCKHLQD Q (66) Vdq, Wdq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PUNPCKHLQD Q (66) Vdq, Wdq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PUNPCKHLQD Q (66) Vdq, Wdq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PUNPCKHLQD Q (66) Vdq, Wdq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>MMX UD (Reserved for future use)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HADDPPD (66) Vpd, Wpd HADDPS (F2) Vps, Wps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HSUBPD (66) Vpd, Wpd HSUBPS (F2) Vps, Wps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOVD Ed, Pd MOVD (66) Ed, Vd MOVD (66) Ed, Vd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOVD Pd, Ed MOVDQD (66) Wdq, Vdq MOVDQD (66) Wdq, Vdq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOVD Pd, Ed MOVDQD (66) Wdq, Vdq MOVDQD (66) Wdq, Vdq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOVD Pd, Ed MOVDQD (66) Wdq, Vdq MOVDQD (66) Wdq, Vdq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>--------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>8</td>
<td>010</td>
<td>NO10</td>
<td>B/C/NAE10</td>
<td>AE/NB/NC10</td>
<td>E/Z10</td>
<td>NE/NZ10</td>
<td>BE/NA10</td>
<td>ANBE10</td>
</tr>
<tr>
<td>9</td>
<td>O</td>
<td>NO</td>
<td>B/C/NAE</td>
<td>AE/NB/NC</td>
<td>E/Z</td>
<td>NE/NZ</td>
<td>BE/NA</td>
<td>ANBE</td>
</tr>
<tr>
<td>A</td>
<td>PUSH</td>
<td>POP</td>
<td>CPUID10</td>
<td>BT</td>
<td>SHLD</td>
<td>SHLD</td>
<td>BE/NA</td>
<td>ANBE</td>
</tr>
<tr>
<td>B</td>
<td>CMPXCHG</td>
<td>LSS</td>
<td>CMPSS</td>
<td>CMPPS</td>
<td>MOVNTI</td>
<td>PINSRW</td>
<td>PEXTRW</td>
<td>MOVZX</td>
</tr>
<tr>
<td></td>
<td>Eb, Gb</td>
<td>Mp</td>
<td>Vps, Wps, Ib</td>
<td>Vps, Wps, Ib</td>
<td>Md, Gd1F</td>
<td>W, Ew, Ib</td>
<td>Gw, Vw, Ib1H</td>
<td>Gv, Eb</td>
</tr>
<tr>
<td>C</td>
<td>XADD</td>
<td>XADD</td>
<td>PSRLW</td>
<td>PSRLD</td>
<td>PSRLQ</td>
<td>PADDO</td>
<td>PMULLW</td>
<td>MOVQ</td>
</tr>
<tr>
<td></td>
<td>Eb, Gb</td>
<td>Eb, Gv</td>
<td>Pq, Qq</td>
<td>Vdq, Wdq</td>
<td>Vdq, Wdq</td>
<td>Vdq, Wdq</td>
<td>Vdq, Wdq</td>
<td>(66)</td>
</tr>
<tr>
<td>D</td>
<td>ADDSUBPD</td>
<td>ADDSUBPS</td>
<td>PSRAW</td>
<td>PSRAD</td>
<td>PAVGW</td>
<td>PMULHUW</td>
<td>PMULHW</td>
<td>MOVNTQ</td>
</tr>
<tr>
<td></td>
<td>(66)</td>
<td>(F2)</td>
<td>Vps, Wps</td>
<td>Vps, Wps</td>
<td>Pq, Qq</td>
<td>Vdq, Wdq</td>
<td>Vdq, Wdq</td>
<td>Mq, Vq</td>
</tr>
<tr>
<td>E</td>
<td>LDDQU</td>
<td>LDDQU (F2)</td>
<td>PSSLW</td>
<td>PSLLD</td>
<td>PSLLQ</td>
<td>PMULDOQ</td>
<td>PMADDWD</td>
<td>PSADDIW</td>
</tr>
<tr>
<td></td>
<td>(66)</td>
<td>Vdq, Mdq</td>
<td>Pq, Qq</td>
<td>Pq, Qq</td>
<td>Pq, Qq</td>
<td>Pq, Qq</td>
<td>Pq, Qq</td>
<td>Pq, Qq</td>
</tr>
</tbody>
</table>

**Table A-3. Two-byte Opcode Map (Proceeding Byte is 0FH)**
Table A-3. Two-byte Opcode Map (Proceeding Byte is 0FH)

<table>
<thead>
<tr>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>S1D</td>
<td>NS1D</td>
<td>P/PE1D</td>
<td>NP/PO1D</td>
<td>L/NGE1D</td>
<td>NL/GE1D</td>
<td>NLE/G1D</td>
</tr>
<tr>
<td>9</td>
<td>SETcc, Eb - Byte Set on condition (000)1C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>PUSH</td>
<td>POP</td>
<td>RSM1D</td>
<td>BTS</td>
<td>SHRD</td>
<td>SHRD</td>
<td>IMUL</td>
</tr>
<tr>
<td>B</td>
<td>Grp 101A</td>
<td>Invalid</td>
<td>Opocode1B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>EAX</td>
<td>ECX</td>
<td>EDX</td>
<td>EBX</td>
<td>ESP</td>
<td>EBP</td>
<td>ESI</td>
</tr>
<tr>
<td>D</td>
<td>PSUBUSB</td>
<td>PSUBUSW</td>
<td>PMINUB</td>
<td>PAND</td>
<td>PADUSB</td>
<td>PADDUSB</td>
<td>PMAXUSB</td>
</tr>
<tr>
<td>E</td>
<td>PSUBSB</td>
<td>PSUBSW</td>
<td>PMINSW</td>
<td>POR</td>
<td>PADDSB</td>
<td>PADSWS</td>
<td>PMAXSW</td>
</tr>
<tr>
<td>F</td>
<td>PSUBB</td>
<td>PSUBW</td>
<td>PSUBD</td>
<td>PSUBQ</td>
<td>PADDB</td>
<td>PADDW</td>
<td>PADD</td>
</tr>
</tbody>
</table>

Note: The table entries represent different opcodes and their associated conditions and operands. The conditions and operands are specified in hexadecimal format, and some entries indicate invalid opcodes or specific operations like set, push, pop, and set on condition, among others.
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).

---

<table>
<thead>
<tr>
<th>Opcode Group Mod</th>
<th>Mod 7,6</th>
<th>Encoding of Bits 5,4,3 of the ModR/M Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>010</td>
</tr>
<tr>
<td>80-83</td>
<td></td>
<td>ADD</td>
</tr>
<tr>
<td>C0, C1 reg, imm</td>
<td></td>
<td>ROL</td>
</tr>
<tr>
<td>F6, F7</td>
<td></td>
<td>TEST</td>
</tr>
<tr>
<td>FE</td>
<td></td>
<td>INC</td>
</tr>
<tr>
<td>FF</td>
<td></td>
<td>INC</td>
</tr>
<tr>
<td>OF 00</td>
<td></td>
<td>SLDT</td>
</tr>
<tr>
<td>OF 01</td>
<td></td>
<td>mem</td>
</tr>
<tr>
<td>OF BA</td>
<td></td>
<td>mem</td>
</tr>
<tr>
<td>OF C7</td>
<td></td>
<td>mem</td>
</tr>
</tbody>
</table>

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

**Table A-4. Opcode Extensions for One- and Two-byte Opcodes by Group Number (Contd.)**

<table>
<thead>
<tr>
<th>Group</th>
<th>Mem</th>
<th>Opcode Extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>OF B9</td>
<td>10</td>
<td>mem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11B</td>
</tr>
<tr>
<td>C6</td>
<td>11</td>
<td>mem, 11B MOV Eb, Ib</td>
</tr>
<tr>
<td>C7</td>
<td>11</td>
<td>mem, 11B MOV Ev, Iv</td>
</tr>
<tr>
<td>OF 71</td>
<td>12</td>
<td>mem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSRLW Pq, lb PSRLW (66) Pdq, lb</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSRAW Pq, lb PSRAW (66) Pdq, lb</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSLLW Pq, lb PSLLW (66) Pdq, lb</td>
</tr>
<tr>
<td>OF 72</td>
<td>13</td>
<td>mem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSRLD Pq, lb PSRLD (66) Wdq, lb</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSRAD Pq, lb PSRAD (66) Wdq, lb</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSLLD Pq, lb PSLLD (66) Wdq, lb</td>
</tr>
<tr>
<td>OF 73</td>
<td>14</td>
<td>mem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSRLQ Pq, lb PSRLQ (66) Wdq, lb</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSRLQ (66) Wdq, lb</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSLLQ Pq, lb PSLLQ (66) Wdq, lb</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSLLQ (66) Wdq, lb</td>
</tr>
<tr>
<td>OF AE</td>
<td>15</td>
<td>mem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FXSAVE FXRSTOR LDMXCSR STMXCSR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LFENCE (000) MFENCE (000) SFENCE (000)</td>
</tr>
<tr>
<td>OF 1B</td>
<td>16</td>
<td>mem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PREFETCH-NTA PREFETCH-T0 PREFETCH-T1 PREFETCH-T2</td>
</tr>
</tbody>
</table>

**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.3.5.1. 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.

| 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

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>C</td>
<td>FADD</td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
</tr>
<tr>
<td>D</td>
<td>FCOM</td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
</tr>
<tr>
<td>E</td>
<td>FSUB</td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
</tr>
<tr>
<td>F</td>
<td>FDIV</td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
</tr>
</tbody>
</table>

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>C</td>
<td>FMUL</td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
</tr>
<tr>
<td>D</td>
<td>FCOMP</td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
</tr>
<tr>
<td>E</td>
<td>FSUBR</td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
</tr>
<tr>
<td>F</td>
<td>FDIVR</td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte (refer to Figure A-1)</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD single-real</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FST single-real</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTP single-real</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDENV 14/28 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDCW 2 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FNSTENV 14/28 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FNSTCW 2 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>C</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FLD</td>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
</tr>
<tr>
<td>D</td>
<td>FNOP</td>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
</tr>
<tr>
<td>E</td>
<td>FCHS</td>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
</tr>
<tr>
<td>F</td>
<td>F2XM1</td>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte (refer to Figure A-1)</th>
<th>FF</th>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
<th>07</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIADD</td>
<td>dword-integer</td>
<td>FIMUL</td>
<td>dword-integer</td>
<td>FICOM</td>
<td>dword-integer</td>
<td>FICOMP</td>
<td>dword-integer</td>
<td>FISUB</td>
<td>dword-integer</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td>FCMOVB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>FCMOVBE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table A-11. DA Opcode Map When ModR/M Byte is Outside 00H to BFH (Continued)

<table>
<thead>
<tr>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td>FCMOVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>FCMOVU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>FUCOMPP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte (refer to Figure A-1)</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILD dword-integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FISTTP dword-integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FISTP dword-integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLD extended-real</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTP extended-real</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>FCMOVNB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>FCMOVNBE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>FNCLEX</td>
<td>FINIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td>FCOMI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte (refer to Figure A-1)</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD (double-real)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMUL (double-real)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCOM (double-real)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCOM (double-real)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSUB (double-real)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSUBR (double-real)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDIV (double-real)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDIVR (double-real)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td>FADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
<td>ST(6),ST(0)</td>
<td>ST(7),ST(0)</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
<td>ST(6),ST(0)</td>
<td>ST(7),ST(0)</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
<td>ST(6),ST(0)</td>
<td>ST(7),ST(0)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td>FMUL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
<td>ST(6),ST(0)</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
<td>ST(6),ST(0)</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
<td>ST(6),ST(0)</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte (refer to Figure A-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00B</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>FFREE</td>
<td>ST(0)</td>
<td>ST(1)</td>
<td>ST(2)</td>
<td>ST(3)</td>
<td>ST(4)</td>
<td>ST(5)</td>
<td>ST(6)</td>
</tr>
<tr>
<td>D</td>
<td>FST</td>
<td>ST(0)</td>
<td>ST(1)</td>
<td>ST(2)</td>
<td>ST(3)</td>
<td>ST(4)</td>
<td>ST(5)</td>
<td>ST(6)</td>
</tr>
<tr>
<td>E</td>
<td>FUCOM</td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
<td>ST(6),ST(0)</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte (refer to Figure A-1)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>000B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>FADD</td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>FSUB</td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
</tr>
<tr>
<td>F</td>
<td>FDIV</td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>FMUL</td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
</tr>
<tr>
<td>D</td>
<td>FCOM</td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
</tr>
<tr>
<td>E</td>
<td>FSUBP</td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
</tr>
<tr>
<td>F</td>
<td>FDIVP</td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
</tr>
</tbody>
</table>

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) select the instruction.

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

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILD</td>
<td>word-integer</td>
<td>FISTTP</td>
<td>word-integer</td>
<td>FISTP</td>
<td>word-integer</td>
<td>FBLD</td>
<td>packed-BCD</td>
<td>FILD</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>FSTSW</td>
<td>AX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>FCOMIP</td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
<td>ST(0),ST(6)</td>
</tr>
</tbody>
</table>

**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.
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).

![Figure B-1. General Machine Instruction Format](image)

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

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

### 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

<table>
<thead>
<tr>
<th>reg Field</th>
<th>Register Selected during 16-Bit Data Operations</th>
<th>Register Selected during 32-Bit Data Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AX</td>
<td>EAX</td>
</tr>
<tr>
<td>001</td>
<td>CX</td>
<td>ECX</td>
</tr>
<tr>
<td>010</td>
<td>DX</td>
<td>EDX</td>
</tr>
<tr>
<td>011</td>
<td>BX</td>
<td>EBX</td>
</tr>
<tr>
<td>100</td>
<td>SP</td>
<td>ESP</td>
</tr>
<tr>
<td>101</td>
<td>BP</td>
<td>EBP</td>
</tr>
<tr>
<td>110</td>
<td>SI</td>
<td>ESI</td>
</tr>
<tr>
<td>111</td>
<td>DI</td>
<td>EDI</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>w Bit</th>
<th>Operand Size When Operand-Size Attribute is 16 Bits</th>
<th>Operand Size When Operand-Size Attribute is 32 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8 Bits</td>
<td>8 Bits</td>
</tr>
<tr>
<td>1</td>
<td>16 Bits</td>
<td>32 Bits</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>s</th>
<th>Effect on 8-Bit Immediate Data</th>
<th>Effect on 16- or 32-Bit Immediate Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>Sign-extend to fill 16-bit or 32-bit destination</td>
<td>None</td>
</tr>
</tbody>
</table>
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).

* Do not use reserved encodings.

<table>
<thead>
<tr>
<th>2-Bit sreg2 Field</th>
<th>Segment Register Selected</th>
<th>3-Bit sreg3 Field</th>
<th>Segment Register Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ES</td>
<td>000</td>
<td>ES</td>
</tr>
<tr>
<td>01</td>
<td>CS</td>
<td>001</td>
<td>CS</td>
</tr>
<tr>
<td>10</td>
<td>SS</td>
<td>010</td>
<td>SS</td>
</tr>
<tr>
<td>11</td>
<td>DS</td>
<td>011</td>
<td>DS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>FS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>101</td>
<td>GS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>110</td>
<td>Reserved*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>111</td>
<td>Reserved*</td>
</tr>
</tbody>
</table>

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.

* Do not use reserved encodings.

<table>
<thead>
<tr>
<th>eee</th>
<th>Control Register</th>
<th>Debug Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>CR0</td>
<td>DR0</td>
</tr>
<tr>
<td>001</td>
<td>Reserved*</td>
<td>DR1</td>
</tr>
<tr>
<td>010</td>
<td>CR2</td>
<td>DR2</td>
</tr>
<tr>
<td>011</td>
<td>CR3</td>
<td>DR3</td>
</tr>
<tr>
<td>100</td>
<td>CR4</td>
<td>Reserved*</td>
</tr>
<tr>
<td>101</td>
<td>Reserved*</td>
<td>Reserved*</td>
</tr>
<tr>
<td>110</td>
<td>Reserved*</td>
<td>DR6</td>
</tr>
<tr>
<td>111</td>
<td>Reserved*</td>
<td>DR7</td>
</tr>
</tbody>
</table>

* Do not use reserved encodings.
INSTRUCTION FORMATS AND 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

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

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.
INSTRUCTION FORMATS AND ENCODINGS

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A value of 11B in bits 7 and 6 of the ModR/M byte is reserved.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA – ASCII Adjust after Addition</td>
<td>0011 0111</td>
</tr>
<tr>
<td>AAD – ASCII Adjust AX before Division</td>
<td>1101 0101 : 0000 1010</td>
</tr>
<tr>
<td>AAM – ASCII Adjust AX after Multiply</td>
<td>1101 0100 : 0000 1010</td>
</tr>
<tr>
<td>AAS – ASCII Adjust AL after Subtraction</td>
<td>0011 1111</td>
</tr>
<tr>
<td>ADC – ADD with Carry</td>
<td>0001 000w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0001 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0001 001w : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0001 000w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 010 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0001 010w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 010 r/m : immediate data</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>ADD – Add</strong></td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0000 000w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0000 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0000 001w : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0000 000w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 000 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0000 010w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 000 r/m : immediate data</td>
</tr>
<tr>
<td><strong>AND – Logical AND</strong></td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0010 000w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0010 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0010 001w : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0010 000w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 100 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0010 010w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 100 r/m : immediate data</td>
</tr>
<tr>
<td><strong>ARPL – Adjust RPL Field of Selector</strong></td>
<td></td>
</tr>
<tr>
<td>from register</td>
<td>0110 0011 : 11 reg1 reg2</td>
</tr>
<tr>
<td>from memory</td>
<td>0110 0011 : mod reg r/m</td>
</tr>
<tr>
<td><strong>BOUND – Check Array Against Bounds</strong></td>
<td>0110 0010 : modA reg r/m</td>
</tr>
<tr>
<td><strong>BSF – Bit Scan Forward</strong></td>
<td></td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1011 1100 : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory, register</td>
<td>0000 1111 : 1011 1100 : mod reg r/m</td>
</tr>
<tr>
<td><strong>BSR – Bit Scan Reverse</strong></td>
<td></td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1011 1101 : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory, register</td>
<td>0000 1111 : 1011 1101 : mod reg r/m</td>
</tr>
<tr>
<td><strong>BSWAP – Byte Swap</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 1111 : 1100 1 reg</td>
</tr>
<tr>
<td><strong>BT – Bit Test</strong></td>
<td></td>
</tr>
<tr>
<td>register, immediate</td>
<td>0000 1111 : 1011 1010 : 11 100 reg: imm8 data</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0000 1111 : 1011 1010 : mod 100 r/m : imm8 data</td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1010 0011 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory, reg</td>
<td>0000 1111 : 1010 0011 : mod reg r/m</td>
</tr>
</tbody>
</table>
### Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BTC – Bit Test and Complement</strong></td>
<td>0000 1111 : 1011 1010 : 11 111 reg : imm8 data</td>
</tr>
<tr>
<td>register, immediate</td>
<td>0000 1111 : 1011 1010 : mod 111 r/m : imm8 data</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0000 1111 : 1011 1011 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory, reg</td>
<td>0000 1111 : 1011 1011 : mod reg r/m</td>
</tr>
<tr>
<td><strong>BTR – Bit Test and Reset</strong></td>
<td>0000 1111 : 1011 1010 : 11 110 reg : imm8 data</td>
</tr>
<tr>
<td>register, immediate</td>
<td>0000 1111 : 1011 1010 : mod 110 r/m : imm8 data</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0000 1111 : 1011 0011 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory, reg</td>
<td>0000 1111 : 1011 0011 : mod reg r/m</td>
</tr>
<tr>
<td><strong>BTS – Bit Test and Set</strong></td>
<td>0000 1111 : 1011 1010 : 11 101 reg : imm8 data</td>
</tr>
<tr>
<td>register, immediate</td>
<td>0000 1111 : 1011 1010 : mod 101 r/m : imm8 data</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0000 1111 : 1010 1011 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory, reg</td>
<td>0000 1111 : 1010 1011 : mod reg r/m</td>
</tr>
<tr>
<td><strong>CALL – Call Procedure (in same segment)</strong></td>
<td>1110 1000 : full displacement</td>
</tr>
<tr>
<td>direct</td>
<td>1111 1111 : 11 010 reg</td>
</tr>
<tr>
<td>memory indirect</td>
<td>1111 1111 : mod 010 r/m</td>
</tr>
<tr>
<td><strong>CALL – Call Procedure (in other segment)</strong></td>
<td>1001 1010 : unsigned full offset, selector</td>
</tr>
<tr>
<td>direct</td>
<td>1111 1111 : mod 011 r/m</td>
</tr>
<tr>
<td>indirect</td>
<td>1001 1000</td>
</tr>
<tr>
<td><strong>CBW – Convert Byte to Word</strong></td>
<td>1111 1000</td>
</tr>
<tr>
<td><strong>CDQ – Convert Doubleword to Qword</strong></td>
<td>1111 1001</td>
</tr>
<tr>
<td><strong>CLC – Clear Carry Flag</strong></td>
<td>1111 1000</td>
</tr>
<tr>
<td><strong>CLD – Clear Direction Flag</strong></td>
<td>1111 1100</td>
</tr>
<tr>
<td><strong>CLI – Clear Interrupt Flag</strong></td>
<td>1111 1010</td>
</tr>
<tr>
<td><strong>CLTS – Clear Task-Switched Flag in CR0</strong></td>
<td>0000 1111 : 0000 0110</td>
</tr>
<tr>
<td><strong>CMC – Complement Carry Flag</strong></td>
<td>1111 0101</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>CMP – Compare Two Operands</strong></td>
<td></td>
</tr>
<tr>
<td>register1 with register2</td>
<td>0011 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 with register1</td>
<td>0011 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory with register</td>
<td>0011 100w : mod reg r/m</td>
</tr>
<tr>
<td>register with memory</td>
<td>0011 101w : mod reg r/m</td>
</tr>
<tr>
<td>immediate with register</td>
<td>1000 00sw : 11 111 reg : immediate data</td>
</tr>
<tr>
<td>immediate with AL, AX, or EAX</td>
<td>0011 110w : immediate data</td>
</tr>
<tr>
<td>immediate with memory</td>
<td>1000 00sw : mod 111 r/m : immediate data</td>
</tr>
<tr>
<td><strong>CMPS/CMPSB/CMPSW/CMPSD – Compare String Operands</strong></td>
<td>1010 011w</td>
</tr>
<tr>
<td><strong>CMPXCHG – Compare and Exchange</strong></td>
<td></td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1011 000w : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory, register</td>
<td>0000 1111 : 1011 000w : mod reg r/m</td>
</tr>
<tr>
<td><strong>CPUID – CPU Identification</strong></td>
<td>0000 1111 : 1010 0010</td>
</tr>
<tr>
<td><strong>CWD – Convert Word to Doubleword</strong></td>
<td>1001 1001</td>
</tr>
<tr>
<td><strong>CWDE – Convert Word to Doubleword</strong></td>
<td>1001 1000</td>
</tr>
<tr>
<td><strong>DAA – Decimal Adjust AL after Addition</strong></td>
<td>0010 0111</td>
</tr>
<tr>
<td><strong>DAS – Decimal Adjust AL after Subtraction</strong></td>
<td>0010 1111</td>
</tr>
<tr>
<td><strong>DEC – Decrement by 1</strong></td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>1111 111w : 11 001 reg</td>
</tr>
<tr>
<td>register (alternate encoding)</td>
<td>0100 1 reg</td>
</tr>
<tr>
<td>memory</td>
<td>1111 111w : mod 001 r/m</td>
</tr>
<tr>
<td><strong>DIV – Unsigned Divide</strong></td>
<td></td>
</tr>
<tr>
<td>AL, AX, or EAX by register</td>
<td>1111 011w : 11 110 reg</td>
</tr>
<tr>
<td>AL, AX, or EAX by memory</td>
<td>1111 011w : mod 110 r/m</td>
</tr>
<tr>
<td><strong>ENTER – Make Stack Frame for High Level Procedure</strong></td>
<td>1100 1000 : 16-bit displacement : 8-bit level (L)</td>
</tr>
<tr>
<td><strong>HLT – Halt</strong></td>
<td>1111 0100</td>
</tr>
<tr>
<td><strong>IDIV – Signed Divide</strong></td>
<td></td>
</tr>
<tr>
<td>AL, AX, or EAX by register</td>
<td>1111 011w : 11 111 reg</td>
</tr>
<tr>
<td>AL, AX, or EAX by memory</td>
<td>1111 011w : mod 111 r/m</td>
</tr>
</tbody>
</table>
### Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IMUL – Signed Multiply</strong></td>
<td></td>
</tr>
<tr>
<td>AL, AX, or EAX with register</td>
<td>1111 011w : 11 101 reg</td>
</tr>
<tr>
<td>AL, AX, or EAX with memory</td>
<td>1111 011w : mod 101 reg</td>
</tr>
<tr>
<td>register1 with register2</td>
<td>0000 1111 : 1010 1111 : 11 : reg1 reg2</td>
</tr>
<tr>
<td>register with memory</td>
<td>0000 1111 : 1010 1111 : mod reg r/m</td>
</tr>
<tr>
<td>register1 with immediate to register2</td>
<td>0110 10s1 : 11 reg1 reg2 : immediate data</td>
</tr>
<tr>
<td>memory with immediate to register</td>
<td>0110 10s1 : mod reg r/m : immediate data</td>
</tr>
<tr>
<td><strong>IN – Input From Port</strong></td>
<td></td>
</tr>
<tr>
<td>fixed port</td>
<td>1110 010w : port number</td>
</tr>
<tr>
<td>variable port</td>
<td>1110 110w</td>
</tr>
<tr>
<td><strong>INC – Increment by 1</strong></td>
<td></td>
</tr>
<tr>
<td>reg</td>
<td>1111 111w : 11 000 reg</td>
</tr>
<tr>
<td>reg (alternate encoding)</td>
<td>0100 0 reg</td>
</tr>
<tr>
<td>memory</td>
<td>1111 111w : mod 000 r/m</td>
</tr>
<tr>
<td><strong>INS – Input from DX Port</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0110 110w</td>
</tr>
<tr>
<td><strong>INT n – Interrupt Type n</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1100 1101 : type</td>
</tr>
<tr>
<td><strong>INT – Single-Step Interrupt 3</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1100 1100</td>
</tr>
<tr>
<td><strong>INTO – Interrupt 4 on Overflow</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1100 1110</td>
</tr>
<tr>
<td><strong>INVD – Invalidate Cache</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 1111 : 0000 1000</td>
</tr>
<tr>
<td><strong>INVLPG – Invalidate TLB Entry</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 1111 : 0000 0001 : mod 111 r/m</td>
</tr>
<tr>
<td><strong>IRET/IRETD – Interrupt Return</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1100 1111</td>
</tr>
<tr>
<td><strong>Jcc – Jump if Condition is Met</strong></td>
<td></td>
</tr>
<tr>
<td>8-bit displacement</td>
<td>0111 tttn : 8-bit displacement</td>
</tr>
<tr>
<td>full displacement</td>
<td>0000 1111 : 1000 tttn : full displacement</td>
</tr>
<tr>
<td><strong>JCXZ/JECXZ – Jump on CX/ECX Zero</strong></td>
<td></td>
</tr>
<tr>
<td>Address-size prefix differentiates JCXZ and JECXZ</td>
<td>1110 0011 : 8-bit displacement</td>
</tr>
<tr>
<td><strong>JMP – Unconditional Jump (to same segment)</strong></td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>1110 1011 : 8-bit displacement</td>
</tr>
<tr>
<td>direct</td>
<td>1110 1001 : full displacement</td>
</tr>
<tr>
<td>register indirect</td>
<td>1111 1111 : 11 100 reg</td>
</tr>
<tr>
<td>memory indirect</td>
<td>1111 1111 : mod 100 r/m</td>
</tr>
</tbody>
</table>
### Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>JMP – Unconditional Jump (to other segment)</strong></td>
<td>1110 1010: unsigned full offset, selector</td>
</tr>
<tr>
<td>direct intersegment</td>
<td>1111 1111: mod 101 r/m</td>
</tr>
<tr>
<td>indirect intersegment</td>
<td>1001 1111</td>
</tr>
<tr>
<td><strong>LAHF – Load Flags into AHRegister</strong></td>
<td>1110 1010: unsigned full offset, selector</td>
</tr>
<tr>
<td><strong>LAR – Load Access Rights Byte</strong></td>
<td>1110 1010: unsigned full offset, selector</td>
</tr>
<tr>
<td>from register</td>
<td>1111 0000: 0000 0001: 11 reg1 reg2</td>
</tr>
<tr>
<td>from memory</td>
<td>1111 1111: 0000 0010: mod reg r/m</td>
</tr>
<tr>
<td><strong>LDS – Load Pointer to DS</strong></td>
<td>1100 0101: modA reg r/m</td>
</tr>
<tr>
<td><strong>LEA – Load Effective Address</strong></td>
<td>1110 0001: modA reg r/m</td>
</tr>
<tr>
<td><strong>LEAVE – High Level Procedure Exit</strong></td>
<td>1001 0000: modA reg r/m</td>
</tr>
<tr>
<td><strong>LES – Load Pointer to ES</strong></td>
<td>1110 1010: modA reg r/m</td>
</tr>
<tr>
<td><strong>LFS – Load Pointer to FS</strong></td>
<td>1110 0101: modA reg r/m</td>
</tr>
<tr>
<td><strong>LGDT – Load Global Descriptor Table Register</strong></td>
<td>0000 1111: 0000 0001: modA 010 r/m</td>
</tr>
<tr>
<td><strong>LGS – Load Pointer to GS</strong></td>
<td>0000 1111: 0000 0001: modA reg r/m</td>
</tr>
<tr>
<td><strong>LIDT – Load Interrupt Descriptor Table Register</strong></td>
<td>0000 1111: 0000 0001: modA 011 r/m</td>
</tr>
<tr>
<td><strong>LLDT – Load Local Descriptor Table Register</strong></td>
<td>0000 1111: 0000 0001: modA 010 r/m</td>
</tr>
<tr>
<td>from register</td>
<td>0000 1111: 0000 0000: 11 010 reg</td>
</tr>
<tr>
<td>from memory</td>
<td>0000 1111: 0000 0000: mod 010 r/m</td>
</tr>
<tr>
<td><strong>LMSW – Load Machine Status Word</strong></td>
<td>1111 0000</td>
</tr>
<tr>
<td>from register</td>
<td>1110 0010: 8-bit displacement</td>
</tr>
<tr>
<td>from memory</td>
<td>1110 0000: 8-bit displacement</td>
</tr>
<tr>
<td><strong>LOCK – Assert LOCK# Signal Prefix</strong></td>
<td>1110 0010: 8-bit displacement</td>
</tr>
<tr>
<td><strong>LODS/LODSB/LODSW/LODSD – Load String Operand</strong></td>
<td>1110 0001: 8-bit displacement</td>
</tr>
<tr>
<td><strong>LOOP – Loop Count</strong></td>
<td>1110 0010: 8-bit displacement</td>
</tr>
<tr>
<td><strong>LOOPZ/LOOPE – Loop Count while Zero/Equal</strong></td>
<td>1110 0011: 8-bit displacement</td>
</tr>
<tr>
<td><strong>LOOPNZ/LOOPNE – Loop Count while not Zero/Equal</strong></td>
<td>1110 0010: 8-bit displacement</td>
</tr>
<tr>
<td><strong>LSL – Load Segment Limit</strong></td>
<td>1110 0010: 8-bit displacement</td>
</tr>
<tr>
<td>from register</td>
<td>1110 0010: 8-bit displacement</td>
</tr>
<tr>
<td>from memory</td>
<td>1110 0010: 8-bit displacement</td>
</tr>
<tr>
<td><strong>LSS – Load Pointer to SS</strong></td>
<td>0000 1111: 1011 0010: modA reg r/m</td>
</tr>
</tbody>
</table>
### Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LTR – Load Task Register</strong></td>
<td></td>
</tr>
<tr>
<td>from register</td>
<td>0000 1111 : 0000 0000 : 11 011 reg</td>
</tr>
<tr>
<td>from memory</td>
<td>0000 1111 : 0000 0000 : mod 011 r/m</td>
</tr>
<tr>
<td><strong>MOV – Move Data</strong></td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>1000 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>1000 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to reg</td>
<td>1000 101w : mod reg r/m</td>
</tr>
<tr>
<td>reg to memory</td>
<td>1000 100w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1100 011w : 11 000 reg : immediate data</td>
</tr>
<tr>
<td>immediate to register (alternate encoding)</td>
<td>1011 w reg : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1100 011w : mod 000 r/m : immediate data</td>
</tr>
<tr>
<td>memory to AL, AX, or EAX</td>
<td>1010 000w : full displacement</td>
</tr>
<tr>
<td>AL, AX, or EAX to memory</td>
<td>1010 001w : full displacement</td>
</tr>
<tr>
<td><strong>MOV – Move to/from Control Registers</strong></td>
<td></td>
</tr>
<tr>
<td>CR0 from register</td>
<td>0000 1111 : 0010 0010 : 11 000 reg</td>
</tr>
<tr>
<td>CR2 from register</td>
<td>0000 1111 : 0010 0010 : 11 010reg</td>
</tr>
<tr>
<td>CR3 from register</td>
<td>0000 1111 : 0010 0010 : 11 011 reg</td>
</tr>
<tr>
<td>CR4 from register</td>
<td>0000 1111 : 0010 0010 : 11 100 reg</td>
</tr>
<tr>
<td>register from CR0-CR4</td>
<td>0000 1111 : 0010 0000 : 11 eee reg</td>
</tr>
<tr>
<td><strong>MOV – Move to/from Debug Registers</strong></td>
<td></td>
</tr>
<tr>
<td>DR0-DR3 from register</td>
<td>0000 1111 : 0010 0011 : 11 eee reg</td>
</tr>
<tr>
<td>DR4-DR5 from register</td>
<td>0000 1111 : 0010 0011 : 11 eee reg</td>
</tr>
<tr>
<td>DR6-DR7 from register</td>
<td>0000 1111 : 0010 0011 : 11 eee reg</td>
</tr>
<tr>
<td>register from DR6-DR7</td>
<td>0000 1111 : 0010 0001 : 11 eee reg</td>
</tr>
<tr>
<td>register from DR4-DR5</td>
<td>0000 1111 : 0010 0001 : 11 eee reg</td>
</tr>
<tr>
<td>register from DR0-DR3</td>
<td>0000 1111 : 0010 0001 : 11 eee reg</td>
</tr>
<tr>
<td><strong>MOV – Move to/from Segment Registers</strong></td>
<td></td>
</tr>
<tr>
<td>register to segment register</td>
<td>1000 1110 : 11 sreg3 reg</td>
</tr>
<tr>
<td>register to SS</td>
<td>1000 1110 : 11 sreg3 reg</td>
</tr>
<tr>
<td>memory to segment reg</td>
<td>1000 1110 : mod sreg3 r/m</td>
</tr>
<tr>
<td>memory to SS</td>
<td>1000 1110 : mod sreg3 r/m</td>
</tr>
<tr>
<td>segment register to register</td>
<td>1000 1100 : 11 sreg3 reg</td>
</tr>
<tr>
<td>segment register to memory</td>
<td>1000 1100 : mod sreg3 r/m</td>
</tr>
</tbody>
</table>
Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV/S/MOVSB/MOVSW/MOVSD – Move Data from String to String</td>
<td>1010 010w</td>
</tr>
<tr>
<td>MOVSX – Move with Sign-Extend</td>
<td>0000 1111 : 1011 111w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td></td>
</tr>
<tr>
<td>memory to reg</td>
<td>0000 1111 : 1011 111w : mod reg r/m</td>
</tr>
<tr>
<td>MOVZX – Move with Zero-Extend</td>
<td>0000 1111 : 1011 011w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td></td>
</tr>
<tr>
<td>memory to register</td>
<td>0000 1111 : 1011 011w : mod reg r/m</td>
</tr>
<tr>
<td>MUL – Unsigned Multiply</td>
<td>1111 011w : 11 100 reg</td>
</tr>
<tr>
<td>AL, AX, or EAX with register</td>
<td>1111 011w : 11 reg</td>
</tr>
<tr>
<td>AL, AX, or EAX with memory</td>
<td>1111 011w : mod 100 reg</td>
</tr>
<tr>
<td>NEG – Two’s Complement Negation</td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>1111 011w : 11 011 reg</td>
</tr>
<tr>
<td>memory</td>
<td>1111 011w : mod 011 r/m</td>
</tr>
<tr>
<td>NOP – No Operation</td>
<td>1001 0000</td>
</tr>
<tr>
<td>NOT – One’s Complement Negation</td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>1111 011w : 11 010 reg</td>
</tr>
<tr>
<td>memory</td>
<td>1111 011w : mod 010 r/m</td>
</tr>
<tr>
<td>OR – Logical Inclusive OR</td>
<td></td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0000 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0000 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0000 101w : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0000 100w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 001 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0000 110w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 001 r/m : immediate data</td>
</tr>
<tr>
<td>OUT – Output to Port</td>
<td></td>
</tr>
<tr>
<td>fixed port</td>
<td>1110 011w : port number</td>
</tr>
<tr>
<td>variable port</td>
<td>1110 111w</td>
</tr>
<tr>
<td>OUTS – Output to DX Port</td>
<td>0110 111w</td>
</tr>
</tbody>
</table>
### Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>POP – Pop a Word from the Stack</strong></td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>1000 1111 : 11 000 reg</td>
</tr>
<tr>
<td>register (alternate encoding)</td>
<td>0101 1 reg</td>
</tr>
<tr>
<td>memory</td>
<td>1000 1111 : mod 000 r/m</td>
</tr>
<tr>
<td><strong>POP – Pop a Segment Register from the Stack</strong></td>
<td>Note: CS cannot be sreg2 in this usage.</td>
</tr>
<tr>
<td>segment register DS, ES</td>
<td>000 sreg2 111</td>
</tr>
<tr>
<td>segment register SS</td>
<td>000 sreg2 111</td>
</tr>
<tr>
<td>segment register FS, GS</td>
<td>0000 1111 : 10 sreg3 001</td>
</tr>
<tr>
<td><strong>POPA/POPAD – Pop All General Registers</strong></td>
<td>0110 0001</td>
</tr>
<tr>
<td><strong>POPF/POPFD – Pop Stack into FLAGS or EFLAGS Register</strong></td>
<td>1001 1101</td>
</tr>
<tr>
<td><strong>PUSH – Push Operand onto the Stack</strong></td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>1111 1111 : 11 110 reg</td>
</tr>
<tr>
<td>register (alternate encoding)</td>
<td>0101 0 reg</td>
</tr>
<tr>
<td>memory</td>
<td>1111 1111 : mod 110 r/m</td>
</tr>
<tr>
<td>immediate</td>
<td>0110 10s0 : immediate data</td>
</tr>
<tr>
<td><strong>PUSH – Push Segment Register onto the Stack</strong></td>
<td></td>
</tr>
<tr>
<td>segment register CS,DS,ES,SS</td>
<td>000 sreg2 110</td>
</tr>
<tr>
<td>segment register FS,GS</td>
<td>0000 1111 : 10 sreg3 000</td>
</tr>
<tr>
<td><strong>PUSHA/PUSHAD – Push All General Registers</strong></td>
<td>0110 0000</td>
</tr>
<tr>
<td><strong>PUSHF/PUSHFD – Push Flags Register onto the Stack</strong></td>
<td>1001 1100</td>
</tr>
<tr>
<td><strong>RCL – Rotate thru Carry Left</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>1101 000w : 11 010 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101 000w : mod 010 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101 001w : 11 010 reg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101 001w : mod 010 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 010 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 010 r/m : imm8 data</td>
</tr>
<tr>
<td><strong>RCR – Rotate thru Carry Right</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>1101 000w : 11 011 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101 000w : mod 011 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101 001w : 11 011 reg</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101 001w : mod 011 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 011 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 011 r/m : imm8 data</td>
</tr>
<tr>
<td>RDMSR – Read from Model-Specific Register</td>
<td>0000 1111 : 0011 0010</td>
</tr>
<tr>
<td>RDPMC – Read Performance Monitoring Counters</td>
<td>0000 1111 : 0011 0011</td>
</tr>
<tr>
<td>RDTSC – Read Time-Stamp Counter</td>
<td>0000 1111 : 0011 0001</td>
</tr>
<tr>
<td>REP INS – Input String</td>
<td>1111 0011 : 0110 110w</td>
</tr>
<tr>
<td>REP LODS – Load String</td>
<td>1111 0011 : 1010 110w</td>
</tr>
<tr>
<td>REP MOVS – Move String</td>
<td>1111 0011 : 1010 010w</td>
</tr>
<tr>
<td>REP OUTS – Output String</td>
<td>1111 0011 : 0110 111w</td>
</tr>
<tr>
<td>REP STOS – Store String</td>
<td>1111 0011 : 1010 101w</td>
</tr>
<tr>
<td>REPE CMPS – Compare String</td>
<td>1111 0011 : 1010 011w</td>
</tr>
<tr>
<td>REPE SCAS – Scan String</td>
<td>1111 0011 : 1010 111w</td>
</tr>
<tr>
<td>REPNE CMPS – Compare String</td>
<td>1111 0010 : 1010 011w</td>
</tr>
<tr>
<td>REPNE SCAS – Scan String</td>
<td>1111 0010 : 1010 111w</td>
</tr>
<tr>
<td>RET – Return from Procedure (to same segment) no argument</td>
<td>1100 0011</td>
</tr>
<tr>
<td>RET – Return from Procedure (to other segment) intersegment</td>
<td>1100 1011</td>
</tr>
<tr>
<td>RET – Return from Procedure (to other segment) adding immediate to SP</td>
<td>1100 0010 : 16-bit displacement</td>
</tr>
<tr>
<td>ROL – Rotate Left register by 1</td>
<td>1101 000w : 11 000 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101 000w : mod 000 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101 001w : 11 000 reg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101 001w : mod 000 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 000 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 000 r/m : imm8 data</td>
</tr>
<tr>
<td>ROR – Rotate Right register by 1</td>
<td>1101 000w : 11 001 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101 000w : mod 001 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101 001w : 11 001 reg</td>
</tr>
</tbody>
</table>
### Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory by CL</td>
<td>1101 001w : mod 001 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 001 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 001 r/m : imm8 data</td>
</tr>
<tr>
<td>RSM – Resume from System Management Mode</td>
<td>0000 1111 : 1010 1010</td>
</tr>
<tr>
<td>SAHF – Store AH into Flags</td>
<td>1001 1110</td>
</tr>
<tr>
<td>SAL – Shift Arithmetic Left</td>
<td>same instruction as SHL</td>
</tr>
<tr>
<td>SAR – Shift Arithmetic Right</td>
<td>1101 000w : 11 111 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101 000w : mod 111 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101 001w : 11 111 reg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101 001w : mod 111 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 111 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 111 r/m : imm8 data</td>
</tr>
<tr>
<td>SBB – Integer Subtraction with Borrow</td>
<td>0001 100w : 11 111 reg1 reg2</td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0001 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0001 101w : mod reg r/m</td>
</tr>
<tr>
<td>memory to register</td>
<td>0001 101w : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0001 100w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 011 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0001 110w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 011 r/m : immediate data</td>
</tr>
<tr>
<td>SCAS/SCASB/SCASW/SCASD – Scan String</td>
<td>1010 111w</td>
</tr>
<tr>
<td>SETcc – Byte Set on Condition</td>
<td>0000 1111 : 1001 tttn : 11 000 reg</td>
</tr>
<tr>
<td>register</td>
<td>0000 1111 : 1001 tttn : mod 000 r/m</td>
</tr>
<tr>
<td>memory</td>
<td>0000 1111 : 1001 tttn : mod 000 r/m</td>
</tr>
<tr>
<td>SGDT – Store Global Descriptor Table Register</td>
<td>0000 1111 : 0000 0001 : mod^a 000 r/m</td>
</tr>
<tr>
<td>SHL – Shift Left</td>
<td>1101 000w : 11 100 reg</td>
</tr>
<tr>
<td>register by 1</td>
<td>1101 000w : mod 100 r/m</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101 000w : 11 100 reg</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101 001w : mod 100 r/m</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101 001w : mod 100 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 100 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 100 r/m : imm8 data</td>
</tr>
</tbody>
</table>
### Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SHLD</strong> – Double Precision Shift Left</td>
<td></td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0000 1111 : 1010 0100 : mod reg r/m : imm8</td>
</tr>
<tr>
<td>register by CL</td>
<td>0000 1111 : 1010 0101 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0000 1111 : 1010 0101 : mod reg r/m</td>
</tr>
<tr>
<td><strong>SHR</strong> – Shift Right</td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>1101 000w : 11 101 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101 000w : mod 101 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101 001w : 11 101 reg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101 001w : mod 101 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 101 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 101 r/m : imm8 data</td>
</tr>
<tr>
<td><strong>SHRD</strong> – Double Precision Shift Right</td>
<td></td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0000 1111 : 1010 1100 : mod reg r/m : imm8</td>
</tr>
<tr>
<td>register by CL</td>
<td>0000 1111 : 1010 1101 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0000 1111 : 1010 1101 : mod reg r/m</td>
</tr>
<tr>
<td><strong>SIDT</strong> – Store Interrupt Descriptor Table Register</td>
<td>0000 1111 : 0000 0001 : mod 001 r/m</td>
</tr>
<tr>
<td><strong>SLDT</strong> – Store Local Descriptor Table Register</td>
<td></td>
</tr>
<tr>
<td>to register</td>
<td>0000 1111 : 0000 0000 : 11 000 reg</td>
</tr>
<tr>
<td>to memory</td>
<td>0000 1111 : 0000 0000 : mod 000 r/m</td>
</tr>
<tr>
<td><strong>SMSW</strong> – Store Machine Status Word</td>
<td></td>
</tr>
<tr>
<td>to register</td>
<td>0000 1111 : 0000 0001 : 11 100 reg</td>
</tr>
<tr>
<td>to memory</td>
<td>0000 1111 : 0000 0001 : mod 100 r/m</td>
</tr>
<tr>
<td><strong>STC</strong> – Set Carry Flag</td>
<td>1111 1001</td>
</tr>
<tr>
<td><strong>STD</strong> – Set Direction Flag</td>
<td>1111 1101</td>
</tr>
<tr>
<td><strong>STI</strong> – Set Interrupt Flag</td>
<td>1111 1011</td>
</tr>
<tr>
<td><strong>STOS/STOSB/STOSW/STOSD</strong> – Store String Data</td>
<td>1010 101w</td>
</tr>
<tr>
<td><strong>STR</strong> – Store Task Register</td>
<td></td>
</tr>
<tr>
<td>to register</td>
<td>0000 1111 : 0000 0000 : 11 001 reg</td>
</tr>
<tr>
<td>to memory</td>
<td>0000 1111 : 0000 0000 : mod 001 r/m</td>
</tr>
</tbody>
</table>
Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SUB – Integer Subtraction</strong></td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0010 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0010 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0010 101w : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0010 100w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 101 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0010 110w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 101 r/m : immediate data</td>
</tr>
<tr>
<td><strong>TEST – Logical Compare</strong></td>
<td></td>
</tr>
<tr>
<td>register1 and register2</td>
<td>1000 010w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory and register</td>
<td>1000 010w : mod reg r/m</td>
</tr>
<tr>
<td>immediate and register</td>
<td>1111 011w : 11 000 reg : immediate data</td>
</tr>
<tr>
<td>immediate and AL, AX, or EAX</td>
<td>1010 100w : immediate data</td>
</tr>
<tr>
<td>immediate and memory</td>
<td>1111 011w : mod 000 r/m : immediate data</td>
</tr>
<tr>
<td><strong>UD2 – Undefined instruction</strong></td>
<td>0000 FFFF : 0000 1011</td>
</tr>
<tr>
<td><strong>VERR – Verify a Segment for Reading</strong></td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>0000 1111 : 0000 0000 : 11 100 reg</td>
</tr>
<tr>
<td>memory</td>
<td>0000 1111 : 0000 0000 : mod 100 r/m</td>
</tr>
<tr>
<td><strong>VERW – Verify a Segment for Writing</strong></td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>0000 1111 : 0000 0000 : 11 101 reg</td>
</tr>
<tr>
<td>memory</td>
<td>0000 1111 : 0000 0000 : mod 101 r/m</td>
</tr>
<tr>
<td><strong>WAIT – Wait</strong></td>
<td>1001 1011</td>
</tr>
<tr>
<td><strong>WBINVD – Writeback and Invalidate Data Cache</strong></td>
<td>0000 1111 : 0000 1001</td>
</tr>
<tr>
<td><strong>WRMSR – Write to Model-Specific Register</strong></td>
<td>0000 1111 : 0011 0000</td>
</tr>
<tr>
<td><strong>XADD – Exchange and Add</strong></td>
<td></td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1100 000w : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory, reg</td>
<td>0000 1111 : 1100 000w : mod reg r/m</td>
</tr>
<tr>
<td><strong>XCHG – Exchange Register/Memory with Register</strong></td>
<td></td>
</tr>
<tr>
<td>register1 with register2</td>
<td>1000 011w : 11 reg1 reg2</td>
</tr>
<tr>
<td>AX or EAX with reg</td>
<td>1001 0 reg</td>
</tr>
<tr>
<td>memory with reg</td>
<td>1000 011w : mod reg r/m</td>
</tr>
<tr>
<td><strong>XLAT/XLATB – Table Look-up Translation</strong></td>
<td>1101 0111</td>
</tr>
</tbody>
</table>

Vol. 2B  B-18
**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**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPXCHG8B – Compare and Exchange 8 Bytes memory, register</td>
<td>0000 1111 : 1100 0111 : mod 001 r/m</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>gg</th>
<th>Granularity of Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Packed Bytes</td>
</tr>
<tr>
<td>01</td>
<td>Packed Words</td>
</tr>
<tr>
<td>10</td>
<td>Packed Doublewords</td>
</tr>
<tr>
<td>11</td>
<td>Quadword</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMMS - Empty MMX technology state</td>
<td>0000 1111:01110111</td>
</tr>
<tr>
<td>MOVD - Move doubleword</td>
<td></td>
</tr>
<tr>
<td>reg to mmreg</td>
<td>0000 1111:01101110: 11 mmxreg reg</td>
</tr>
<tr>
<td>reg from mmxreg</td>
<td>0000 1111:011111110: 11 mmxreg reg</td>
</tr>
<tr>
<td>mem to mmxreg</td>
<td>0000 1111:01101110: mod mmxreg r/m</td>
</tr>
<tr>
<td>mem from mmxreg</td>
<td>0000 1111:011111110: mod mmxreg r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-----------------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>MOVQ - Move quadword</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:01101111: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg2 from mmxreg1</td>
<td>0000 1111:01111111: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mem to mmxreg</td>
<td>0000 1111:01101111: mod mmxreg r/m</td>
</tr>
<tr>
<td>mem from mmxreg</td>
<td>0000 1111:01111111: mod mmxreg r/m</td>
</tr>
<tr>
<td>PACKSSDWD&lt;sup&gt;1&lt;/sup&gt; - Pack dword to word data (signed with saturation)</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:01101011: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:011101011: mod mmxreg r/m</td>
</tr>
<tr>
<td>PACKSSWB&lt;sup&gt;1&lt;/sup&gt; - Pack word to byte data (signed with saturation)</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:01100011: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:011100011: mod mmxreg r/m</td>
</tr>
<tr>
<td>PACKUSWB&lt;sup&gt;1&lt;/sup&gt; - Pack word to byte data (unsigned with saturation)</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:01100111: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:011100111: mod mmxreg r/m</td>
</tr>
<tr>
<td>PADD - Add with wrap-around</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:111111111: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:111111111: mod mmxreg r/m</td>
</tr>
<tr>
<td>PADDS - Add signed with saturation</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:111011111: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:111111111: mod mmxreg r/m</td>
</tr>
<tr>
<td>PADDUS - Add unsigned with saturation</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:110111111: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:111111111: mod mmxreg r/m</td>
</tr>
<tr>
<td>PAND - Bitwise And</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:110110111: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:110110111: mod mmxreg r/m</td>
</tr>
<tr>
<td>PANDN - Bitwise AndNot</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:110111111: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:110111111: mod mmxreg r/m</td>
</tr>
</tbody>
</table>
### Table B-14. MMX Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PCMPEQ - Packed compare for equality</strong></td>
<td></td>
</tr>
<tr>
<td>mmxreg1 with mmxreg2</td>
<td>0000 1111:011101gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg with memory</td>
<td>0000 1111:011101gg: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PCMPGT - Packed compare greater (signed)</strong></td>
<td></td>
</tr>
<tr>
<td>mmxreg1 with mmxreg2</td>
<td>0000 1111:011001gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg with memory</td>
<td>0000 1111:011001gg: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PMADDWD - Packed multiply add</strong></td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:11110101: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:11110101: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PMULHUW - Packed multiplication, store high word (unsigned)</strong></td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:11110101: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:11110101: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PMULHW - Packed multiplication, store high word</strong></td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:11110101: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:11110101: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PMULLW - Packed multiplication, store low word</strong></td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:11110101: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:11110101: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>POR - Bitwise Or</strong></td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:11110101: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:11110101: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PSLL2 - Packed shift left logical</strong></td>
<td></td>
</tr>
<tr>
<td>mmxreg1 by mmxreg2</td>
<td>0000 1111:111100gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg by memory</td>
<td>0000 1111:111100gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>mmxreg by immediate</td>
<td>0000 1111:011100gg: 11 110 mmxreg: imm8 data</td>
</tr>
<tr>
<td><strong>PSRA2 - Packed shift right arithmetic</strong></td>
<td></td>
</tr>
<tr>
<td>mmxreg1 by mmxreg2</td>
<td>0000 1111:111100gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg by memory</td>
<td>0000 1111:111100gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>mmxreg by immediate</td>
<td>0000 1111:011100gg: 11 100 mmxreg: imm8 data</td>
</tr>
</tbody>
</table>
Table B-14. MMX Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PSRL² - Packed shift right logical</strong></td>
<td></td>
</tr>
<tr>
<td>mmxreg1 by mmxreg2</td>
<td>0000 1111:110100gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg by memory</td>
<td>0000 1111:110100gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>mmxreg by immediate</td>
<td>0000 1111:011100gg: 11 010 mmxreg: imm8 data</td>
</tr>
<tr>
<td><strong>PSUB - Subtract with wrap-around</strong></td>
<td></td>
</tr>
<tr>
<td>mmxreg2 from mmxreg1</td>
<td>0000 1111:111110gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory from mmxreg</td>
<td>0000 1111:111110gg: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PSUBS - Subtract signed with saturation</strong></td>
<td></td>
</tr>
<tr>
<td>mmxreg2 from mmxreg1</td>
<td>0000 1111:111010gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory from mmxreg</td>
<td>0000 1111:111010gg: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PSUBUS - Subtract unsigned with saturation</strong></td>
<td></td>
</tr>
<tr>
<td>mmxreg2 from mmxreg1</td>
<td>0000 1111:110110gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory from mmxreg</td>
<td>0000 1111:110110gg: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PUNPCKH - Unpack high data to next larger type</strong></td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:011010gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:011010gg: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PUNPCKL - Unpack low data to next larger type</strong></td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:011000gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:011000gg: mod mmxreg r/m</td>
</tr>
<tr>
<td><strong>PXOR - Bitwise Xor</strong></td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:11101111: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:11101111: mod mmxreg r/m</td>
</tr>
</tbody>
</table>

**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.
INSTRUCTION FORMATS AND ENCODINGS

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>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMOVcc – Conditional Move</td>
<td>0000 1111: 1010 tttn: 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0000 1111: 0100 tttn: 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0000 1111: 0100 tttn: mod reg r/m</td>
</tr>
<tr>
<td>FCOMVcc – Conditional Move on EFLAG</td>
<td>11011 011: 11 110 ST(i)</td>
</tr>
<tr>
<td>Register Condition Codes</td>
<td>11011 010: 11 000 ST(i)</td>
</tr>
<tr>
<td>move if below (B)</td>
<td>11011 010: 11 001 ST(i)</td>
</tr>
<tr>
<td>move if equal (E)</td>
<td>11011 010: 11 010 ST(i)</td>
</tr>
<tr>
<td>move if below or equal (BE)</td>
<td>11011 010: 11 011 ST(i)</td>
</tr>
<tr>
<td>move if unordered (U)</td>
<td>11011 010: 11 011 ST(i)</td>
</tr>
<tr>
<td>move if not below (NB)</td>
<td>11011 011: 11 000 ST(i)</td>
</tr>
<tr>
<td>move if not equal (NE)</td>
<td>11011 011: 11 001 ST(i)</td>
</tr>
<tr>
<td>move if not below or equal (NBE)</td>
<td>11011 011: 11 010 ST(i)</td>
</tr>
<tr>
<td>move if not unordered (NU)</td>
<td>11011 011: 11 011 ST(i)</td>
</tr>
<tr>
<td>FCOMI – Compare Real and Set EFLAGS</td>
<td>11011 011: 11 110 ST(i)</td>
</tr>
<tr>
<td>FXRSTOR—Restore x87 FPU, MMX, SSE, and SSE2 State</td>
<td>00001111: 10101110: modA 001 r/m</td>
</tr>
<tr>
<td>FXSAVE—Save x87 FPU, MMX, SSE, and SSE2 State</td>
<td>00001111: 10101110: modA 000 r/m</td>
</tr>
<tr>
<td>SYSENTER—Fast System Call</td>
<td>00001111:00110100</td>
</tr>
<tr>
<td>SYSEXIT—Fast Return from Fast System Call</td>
<td>00001111:00110101</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADDPS—Add Packed Single-Precision Floating-Point Values</strong></td>
<td>xmmreg to xmmreg: 00001111:01010100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg: 00001111:01010100: mod xmmreg r/m</td>
<td></td>
</tr>
<tr>
<td><strong>ADDSS—Add Scalar Single-Precision Floating-Point Values</strong></td>
<td>xmmreg to xmmreg: 11110011:00001111:01011000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg: 11110011:00001111:01011000: mod xmmreg r/m</td>
<td></td>
</tr>
<tr>
<td><strong>ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values</strong></td>
<td>xmmreg to xmmreg: 00001111:01010101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg: 00001111:01010101: mod xmmreg r/m</td>
<td></td>
</tr>
<tr>
<td><strong>ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values</strong></td>
<td>xmmreg to xmmreg: 00001111:01010100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg: 00001111:01010100: mod xmmreg r/m</td>
<td></td>
</tr>
<tr>
<td><strong>CMPPS—Compare Packed Single-Precision Floating-Point Values</strong></td>
<td>xmmreg to xmmreg, imm8: 00001111:11000010:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8: 00001111:11000010: mod xmmreg r/m: imm8</td>
<td></td>
</tr>
<tr>
<td><strong>CMPPS—Compare Scalar Single-Precision Floating-Point Values</strong></td>
<td>xmmreg to xmmreg, imm8: 11110011:00001111:11000010:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8: 11110011:00001111:11000010: mod xmmreg r/m: imm8</td>
<td></td>
</tr>
</tbody>
</table>
### Table B-16. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>00001111:00101111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>00001111:00101111: mod xmmreg r/m</td>
</tr>
<tr>
<td>CVTP2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>mmreg to xmmreg</td>
<td>00001111:00101010:11 mmreg1 mmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>00001111:00101010: mod xmmreg r/m</td>
</tr>
<tr>
<td>CVT2PS2PL—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg to mmreg</td>
<td>00001111:00101101:11 mmreg1 xmmreg1</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>00001111:00101101: mod mmreg r/m</td>
</tr>
<tr>
<td>CVT2S2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>r32 to xmmreg</td>
<td>11110011:00010111:00101010:11 xmmreg r32</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110011:00010111:00101010: mod xmmreg r/m</td>
</tr>
<tr>
<td>CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer</td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>11110011:00010111:00101010:11 r32 xmmreg</td>
</tr>
<tr>
<td>mem to r32</td>
<td>11110011:00010111:00101010: mod r32 r/m</td>
</tr>
<tr>
<td>CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg to mmreg</td>
<td>00001111:00101100:11 mmreg1 xmmreg1</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>00001111:00101100: mod mmreg r/m</td>
</tr>
<tr>
<td>CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer</td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>11110011:00010111:00101100:11 r32 xmmreg1</td>
</tr>
<tr>
<td>mem to r32</td>
<td>11110011:00010111:00101100: mod r32 r/m</td>
</tr>
<tr>
<td>DIVPS—Divide Packed Single-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>00001111:01011101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>00001111:01011110: mod xmmreg r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>DIVSS</strong>—Divide Scalar Single-Precision Floating-Point Values</td>
<td>11110011:00001111:01011110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110011:00001111:01011110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>LDMXCSR</strong>—Load MXCSR Register State</td>
<td>00001111:10101110:modA 010 mem</td>
</tr>
<tr>
<td>m32 to MXCSR</td>
<td>00001111:01011111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MAXPS</strong>—Return Maximum Packed Single-Precision Floating-Point Values</td>
<td>00001111:01011111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>00001111:01011111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MAXSS</strong>—Return Maximum Scalar Double-Precision Floating-Point Value</td>
<td>11110011:00001111:01011111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110011:00001111:01011111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MINPS</strong>—Return Minimum Packed Double-Precision Floating-Point Values</td>
<td>00001111:01011110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>00001111:01011110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MINSS</strong>—Return Minimum Scalar Double-Precision Floating-Point Value</td>
<td>11110011:00001111:01011110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110011:00001111:01011110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVAPS</strong>—Move Aligned Packed Single-Precision Floating-Point Values</td>
<td>00001111:00101000:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>00001111:00101000: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>00001111:00101001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>00001111:00101001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVHLPS</strong>—Move Packed Single-Precision Floating-Point Values High to Low</td>
<td>00001111:00010010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>00001111:00010010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVHPS</strong>—Move High Packed Single-Precision Floating-Point Values</td>
<td>00001111:00010110: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>00001111:00010111: mod xmmreg r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>MOVLP</strong>—Move Low Packed Single-Precision Floating-Point Values mem to xmmreg</td>
<td>00001111:00010010: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg to mem</td>
<td>00001111:00010011: mod xmmreg r/m</td>
</tr>
</tbody>
</table>
| **MOVMSK**—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 |
| **MULP**—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 | |}
<p>| xmmreg to xmmreg | 00001111:01010110:11 xmmreg1 xmmreg2 |
| mem to xmmreg | 00001111:01010110 mod xmmreg r/m |</p>
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values</td>
<td>xmmreg to xmmreg 00001111:01010011:11 xmmreg1 xmmreg2 &lt;br/&gt;mem to xmmreg 00001111:01010011: mod xmmreg r/m</td>
</tr>
<tr>
<td>RCPSS—Compute Reciprocals of Scalar Single-Precision Floating-Point Value</td>
<td>xmmreg to xmmreg 11110011:00001111:01010011:11 xmmreg1 xmmreg2 &lt;br/&gt;mem to xmmreg 11110011:00001111:01010011: mod xmmreg r/m</td>
</tr>
<tr>
<td>RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values</td>
<td>xmmreg to xmmreg 00001111:01010010:11 xmmreg1 xmmreg2 &lt;br/&gt;mem to xmmreg 00001111:01010010: mod xmmreg r/m</td>
</tr>
<tr>
<td>RSQRTSS—Compute Reciprocals of Square Roots of Scalar Single-Precision Floating-Point Value</td>
<td>xmmreg to xmmreg 11110011:00001111:01010010:11 xmmreg1 xmmreg2 &lt;br/&gt;mem to xmmreg 11110011:00001111:01010010: mod xmmreg r/m</td>
</tr>
<tr>
<td>SHUFPS—Shuffle Packed Single-Precision Floating-Point Values</td>
<td>xmmreg to xmmreg, imm8 00001111:11000110:11 xmmreg1 xmmreg2: imm8 &lt;br/&gt;mem to xmmreg, imm8 00001111:11000110: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values</td>
<td>xmmreg to xmmreg 00001111:01010001:11 xmmreg1 xmmreg2 &lt;br/&gt;mem to xmmreg 00001111:01010001: mod xmmreg r/m</td>
</tr>
<tr>
<td>SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value</td>
<td>xmmreg to xmmreg 11110011:00001111:01010001:11 xmmreg1 xmmreg2 &lt;br/&gt;mem to xmmreg 11110011:00001111:01010001: mod xmmreg r/m</td>
</tr>
<tr>
<td>STMXCSR—Store MXCSR Register State</td>
<td>MXCSR to mem 00001111:10101110: mod xmmreg r/m</td>
</tr>
<tr>
<td>SUBPS—Subtract Packed Single-Precision Floating-Point Values</td>
<td>xmmreg to xmmreg 00001111:01011100:11 xmmreg1 xmmreg2 &lt;br/&gt;mem to xmmreg 00001111:01011100: mod xmmreg r/m</td>
</tr>
</tbody>
</table>
Table B-16. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SUBSS</strong>—Subtract Scalar Single-Precision Floating-Point Values</td>
<td>11110011:00001111:01011100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>11110011:00001111:01011100:mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td><strong>UCOMISS</strong>—Unordered Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS</td>
<td>00001111:00101110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>00001111:00101110 mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td><strong>UNPCKHPS</strong>—Unpack and Interleave High Packed Single-Precision Floating-Point Values</td>
<td>00001111:00010101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>00001111:00010101:mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td><strong>UNPCKLPS</strong>—Unpack and Interleave Low Packed Single-Precision Floating-Point Values</td>
<td>00001111:00010100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>00001111:00010100:mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td><strong>XORPS</strong>—Bitwise Logical XOR of Single-Precision Floating-Point Values</td>
<td>00001111:01010111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>00001111:01010111:mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
</tbody>
</table>
### Table B-17. Formats and Encodings of SSE Integer Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PAVGB/PAVGW—Average Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg to mmreg</td>
<td>00001111:11100000:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>00001111:11100011:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td><strong>PEXTRW—Extract Word</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg to reg32, imm8</td>
<td>00001111:11000101:11 r32 mmreg: imm8</td>
</tr>
<tr>
<td><strong>PINSRW - Insert Word</strong></td>
<td></td>
</tr>
<tr>
<td>reg32 to mmreg, imm8</td>
<td>00001111:11000100:11 mmreg r32: imm8</td>
</tr>
<tr>
<td>m16 to mmreg, imm8</td>
<td>00001111:11000100 mod mmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PMAXSW—Maximum of Packed Signed Word Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg to mmreg</td>
<td>00001111:11101110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>00001111:11101110 mod mmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PMAXUB—Maximum of Packed Unsigned Byte Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg to mmreg</td>
<td>00001111:11101110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>00001111:11011110 mod mmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PMINSW—Minimum of Packed Signed Word Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg to mmreg</td>
<td>00001111:11101010:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>00001111:11010110 mod mmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PMINUB—Minimum of Packed Unsigned Byte Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg to mmreg</td>
<td>00001111:11011010:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>00001111:11010110 mod mmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PMOVMSKB - Move Byte Mask To Integer</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg to reg32</td>
<td>00001111:11010111:11 r32 mmreg</td>
</tr>
<tr>
<td><strong>PMULHUW—Multiply Packed Unsigned Integers and Store High Result</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg to mmreg</td>
<td>00001111:11100100:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>00001111:11100100 mod mmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PSADBW—Compute Sum of Absolute Differences</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg to mmreg</td>
<td>00001111:11101110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>00001111:11101110 mod mmreg r/m: imm8</td>
</tr>
</tbody>
</table>
### Table B-17. Formats and Encodings of SSE Integer Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSHUFW—Shuffle Packed Words</td>
<td>00001111:01110000:11 mmreg1 mmreg2: imm8</td>
</tr>
<tr>
<td>mmreg to mmreg, imm8</td>
<td></td>
</tr>
<tr>
<td>mem to mmreg, imm8</td>
<td>00001111:01110000:11 mod mmreg r/m: imm8</td>
</tr>
</tbody>
</table>

Table B-18. Format and Encoding of SSE Cacheability and Memory Ordering Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASKMOVQ—Store Selected Bytes of Quadword</td>
<td>00001111:11110111:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mmreg to mmreg</td>
<td></td>
</tr>
<tr>
<td>MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint</td>
<td>00001111:00101011: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg to mem</td>
<td></td>
</tr>
<tr>
<td>MOVNTQ—Store Quadword Using Non-Temporal Hint</td>
<td>00001111:11100111: mod mmreg r/m</td>
</tr>
<tr>
<td>mmreg to mem</td>
<td></td>
</tr>
<tr>
<td>PREFETCHT0—Prefetch Temporal to All Cache Levels</td>
<td>00001111:11001111: mod mmreg r/m</td>
</tr>
<tr>
<td>PREFETCHT1—Prefetch Temporal to First Level Cache</td>
<td>00001111:00011000:modA 001 mem</td>
</tr>
<tr>
<td>PREFETCHT2—Prefetch Temporal to Second Level Cache</td>
<td>00001111:00011000:modA 010 mem</td>
</tr>
<tr>
<td>PREFETCHNTA—Prefetch Non-Temporal to All Cache Levels</td>
<td>00001111:00011000:modA 011 mem</td>
</tr>
<tr>
<td>SFENCE—Store Fence</td>
<td>00001111:10101110:11 111 000</td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>gg</th>
<th>Granularity of Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Packed Bytes</td>
</tr>
<tr>
<td>01</td>
<td>Packed Words</td>
</tr>
<tr>
<td>10</td>
<td>Packed Doublewords</td>
</tr>
<tr>
<td>11</td>
<td>Quadword</td>
</tr>
</tbody>
</table>

Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPD - Add Packed Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:0101000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:0101000: mod xmmreg r/m</td>
</tr>
<tr>
<td>ADDSD - Add Scalar Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>11110010:00001111:0101000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110010:00001111:0101000: mod xmmreg r/m</td>
</tr>
<tr>
<td>ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:01010101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:01010101: mod xmmreg r/m</td>
</tr>
</tbody>
</table>
### Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ANDPD</strong>—Bitwise Logical AND of Packed Double-Precision Floating-Point Values</td>
<td>01100110:00001111:01010100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:01010100: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:01010100: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CMPPD</strong>—Compare Packed Double-Precision Floating-Point Values</td>
<td>01100110:00001111:11000010:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>xmmreg to xmmreg, imm8</td>
<td>01100110:00001111:11000010: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>01100110:00001111:11000010: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>CMPSD</strong>—Compare Scalar Double-Precision Floating-Point Values</td>
<td>11110010:00001111:11000010:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>xmmreg to xmmreg, imm8</td>
<td>11110010:00001111:11000010: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>11110010:00001111:11000010: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>COMISD</strong>—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS</td>
<td>01100110:00001111:00101111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:00101111: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:00101111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTPI2PD</strong>—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values</td>
<td>01100110:00001111:00101010:11 xmmreg1 mmreg1</td>
</tr>
<tr>
<td>mmreg to xmmreg</td>
<td>01100110:00001111:00101010: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:00101010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTPD2PI</strong>—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</td>
<td>01100110:00001111:00101101:11 mmreg1 xmmreg1</td>
</tr>
<tr>
<td>xmmreg to mmreg</td>
<td>01100110:00001111:00101101: mod mmreg r/m</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>01100110:00001111:00101101: mod mmreg r/m</td>
</tr>
<tr>
<td><strong>CVTSI2SD</strong>—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value</td>
<td>11110010:00001111:00101010:11 xmmreg r32</td>
</tr>
<tr>
<td>r32 to xmmreg1</td>
<td>11110010:00001111:00101010: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110010:00001111:00101010: mod xmmreg r/m</td>
</tr>
</tbody>
</table>
### INSTRUCTION FORMATS AND ENCODINGS

#### Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer</td>
<td>11110011:00001111:01011010:11 r32 xmmreg</td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>11110010:00001111:00101110:00001111:00101101:11 r32 xmmreg</td>
</tr>
<tr>
<td>mem to r32</td>
<td>11110010:00001111:00101100:11 r32 xmmreg</td>
</tr>
<tr>
<td>CVTTPD2PI—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</td>
<td>01100110:00001111:00101111:00101100:11 mmreg xmmreg</td>
</tr>
<tr>
<td>xmmreg to mmreg</td>
<td>01100110:00001111:00101111:00101100:11 mmreg xmmreg</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>01100110:00001111:00101100:11 mmreg r/m</td>
</tr>
<tr>
<td>CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Doubleword Integer</td>
<td>11110011:00001111:01011010:11 r32 xmmreg</td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>11110010:00001111:00101110:00001111:00101101:11 r32 xmmreg</td>
</tr>
<tr>
<td>mem to r32</td>
<td>11110010:00001111:00101100:11 r32 xmmreg</td>
</tr>
<tr>
<td>CVTPD2PS—Convert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values</td>
<td>01100110:00001111:01011010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:01011010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:01011010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>CVTPS2PD—Convert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values</td>
<td>00001111:01011010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>00001111:01011010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>00001111:01011010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value</td>
<td>11110011:00001111:01011010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>11110010:00001111:01011010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110010:00001111:01011010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value</td>
<td>11110011:00001111:01011010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>11110010:00001111:01011010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110010:00001111:01011010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-------------------------------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td><strong>CVTPD2DQ</strong>—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>11110010:00001111:11100110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110010:00001111:11100110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTTPD2DQ</strong>—Convert With Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:11100110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:11100110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTDQ2PD</strong>—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>11110011:00001111:11100110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110011:00001111:11100110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTSPS2DQ</strong>—Convert Single-Precision Floating-Point Values to Packed Doubleword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:01011011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:01011011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTTPS2DQ</strong>—Convert With Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>11110011:00001111:01011011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110011:00001111:01011011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTDQ2PS</strong>—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>00001111:01011011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>00001111:01011011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>DIVPD</strong>—Divide Packed Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:01011110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:01011110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>DIVSD</strong>—Divide Scalar Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>11110010:00001111:01011110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110010:00001111:01011110: mod xmmreg r/m</td>
</tr>
</tbody>
</table>
### Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXPD—Return Maximum Packed Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:01011111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:01011111: mod xmmreg r/m</td>
</tr>
<tr>
<td>MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>11110010:00001111:01011111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110010:00001111:01011111: mod xmmreg r/m</td>
</tr>
<tr>
<td>MINPD—Return Minimum Packed Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:01011101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:01011101: mod xmmreg r/m</td>
</tr>
<tr>
<td>MINSD—Return Minimum Scalar Double-Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>11110010:00001111:01011101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110010:00001111:01011101: mod xmmreg r/m</td>
</tr>
<tr>
<td>MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>01100110:00001111:00101001:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>01100110:00001111:00101001: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>01100110:00001111:00101000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>01100110:00001111:00101000: mod xmmreg r/m</td>
</tr>
<tr>
<td>MOVHPD—Move High Packed Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:00010111: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg to mem</td>
<td>01100110:00001111:00010110: mod xmmreg r/m</td>
</tr>
<tr>
<td>MOVLPD—Move Low Packed Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:00010011: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg to mem</td>
<td>01100110:00001111:00010010: mod xmmreg r/m</td>
</tr>
<tr>
<td>MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask</td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>01100110:00001111:01010000:11 r32 xmmreg</td>
</tr>
</tbody>
</table>
### INSTRUCTION FORMATS AND ENCODINGS

#### Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MOVSD</strong>—Move Scalar Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>11110010:00001111:00010001:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>11110010:00001111:00010001: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>11110010:00001111:00010000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>11110010:00001111:00010000: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVUPD</strong>—Move Unaligned Packed Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>01100110:00001111:00010001:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>01100110:00001111:00010001: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>01100110:00001111:00010000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>01100110:00001111:00010000: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MULPD</strong>—Multiply Packed Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:01011001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:01011001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MULSD</strong>—Multiply Scalar Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>11110010:00001111:01011001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110010:00001111:01011001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>ORPD</strong>—Bitwise Logical OR of Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:01010001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:01010001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>SHUFPD</strong>—Shuffle Packed Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg, imm8</td>
<td>01100110:00001111:01100110:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>01100110:00001111:01100110: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>SQRTPD</strong>—Compute Square Roots of Packed Double-Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:11001110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:11001110: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

Vol. 2B  B-38
### Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
</table>
| SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value | xmmreg to xmmreg: 11110010:00001111:01010001:11: xmmreg1 xmmreg 2  
mem to xmmreg: 11110010:00001111:01010001:01: mod xmmreg r/m |
| SUBPD—Subtract Packed Double-Precision Floating-Point Values | xmmreg to xmmreg: 01100110:00001111:01011100:11: xmmreg1 xmmreg2  
mem to xmmreg: 01100110:00001111:01011100:01: mod xmmreg r/m |
| SUBSD—Subtract Scalar Double-Precision Floating-Point Values | xmmreg to xmmreg: 11110010:00001111:01011100:11: xmmreg1 xmmreg2  
mem to xmmreg: 11110010:00001111:01011100:01: 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: 01100110:00001111:00101110:01: mod xmmreg r/m |
| UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values | xmmreg to xmmreg: 01100110:00001111:01010111:11: xmmreg1 xmmreg2  
mem to xmmreg: 01100110:00001111:01010111:01: 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: 01100110:00001111:00010100:01: 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: 01100110:00001111:01010111:01: mod xmmreg r/m |
### Table B-21. Formats and Encodings of SSE2 Integer Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MOVD - Move Doubleword</strong></td>
<td></td>
</tr>
<tr>
<td>reg to xmmreg</td>
<td>01100110:0000 1111:01101110: 11 xmmreg reg</td>
</tr>
<tr>
<td>reg from xmmreg</td>
<td>01100110:0000 1111:01111110: 11 xmmreg reg</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:0000 1111:01101110: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem from xmmreg</td>
<td>01100110:0000 1111:01111110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVDQA—Move Aligned Double Quadword</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:01101111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:01111111: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem from xmmreg</td>
<td>01100110:00001111:01111111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVDQU—Move Unaligned Double Quadword</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>11110011:00001111:01101111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110011:00001111:01111111: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem from xmmreg</td>
<td>11110011:00001111:01111111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVQ2DQ—Move Quadword from MMX to XMM Register</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg to xmmreg</td>
<td>11110011:00001111:11010110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td><strong>MOVDQ2Q—Move Quadword from XMM to MMX Register</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to mmreg</td>
<td>11110010:00001111:11010110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td><strong>MOVQ - Move Quadword</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>11110011:00001111:01111110: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 from xmmreg1</td>
<td>01100110:00001111:01111111:11010110: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110011:00001111:01111111:01111110: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem from xmmreg</td>
<td>01100110:00001111:11010110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PACKSSDW</strong> - Pack Dword To Word Data (signed with saturation)</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>01100110:0000 1111:01101011: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>01100110:0000 1111:01101011: mod xmmreg r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-----------------------------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PACKSSWB - Pack Word To Byte Data (signed with saturation)</td>
<td>01100110:0000 1111:01100011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>01100110:0000 1111:01100011: mod xmmreg r/m</td>
</tr>
<tr>
<td>PACKUSWB - Pack Word To Byte Data (unsigned with saturation)</td>
<td>01100110:0000 1111:01100111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>01100110:0000 1111:01100111: mod xmmreg r/m</td>
</tr>
<tr>
<td>PADDQ—Add Packed Quadword Integers</td>
<td>00001111:11010100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>00001111:11010100: mod xmmreg r/m</td>
</tr>
<tr>
<td>PADD - Add With Wrap-around</td>
<td>01100110:0000 1111:111111gg:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>01100110:0000 1111:111111gg: mod xmmreg r/m</td>
</tr>
<tr>
<td>PADDSS - Add Signed With Saturation</td>
<td>01100110:0000 1111:111011gg:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>01100110:0000 1111:111011gg: mod xmmreg r/m</td>
</tr>
<tr>
<td>PADDUS - Add Unsigned With Saturation</td>
<td>01100110:0000 1111:110111gg:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>01100110:0000 1111:110111gg: mod xmmreg r/m</td>
</tr>
<tr>
<td>PAND - Bitwise And</td>
<td>01100110:0000 1111:11011011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>01100110:0000 1111:11011011: mod xmmreg r/m</td>
</tr>
<tr>
<td>PANDN - Bitwise AndNot</td>
<td>01100110:0000 1111:11011111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>01100110:0000 1111:11011111: mod xmmreg r/m</td>
</tr>
<tr>
<td>PAVGB—Average Packed Integers</td>
<td>01100110:00001111:11000000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>01100110:00001111:11000000 mod xmmreg r/m</td>
</tr>
</tbody>
</table>
### Table B-21. Formats and Encodings of SSE2 Integer Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PAVGW—Average Packed Integers</strong></td>
<td>01100110:00001111:11100011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td><strong>PCMPEQ - Packed Compare For Equality</strong></td>
<td>01100110:00001111:011101gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg1 with xmmreg2</td>
<td></td>
</tr>
<tr>
<td>xmmreg with memory</td>
<td></td>
</tr>
<tr>
<td><strong>PCMPGT - Packed Compare Greater (signed)</strong></td>
<td>01100110:00001111:011101gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg1 with xmmreg2</td>
<td></td>
</tr>
<tr>
<td>xmmreg with memory</td>
<td></td>
</tr>
<tr>
<td>xmmreg to reg32, imm8</td>
<td></td>
</tr>
<tr>
<td>reg32 to xmmreg, imm8</td>
<td></td>
</tr>
<tr>
<td>m16 to xmmreg, imm8</td>
<td></td>
</tr>
<tr>
<td><strong>PMADDWD - Packed Multiply Add</strong></td>
<td>01100110:00001111:11110101: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td></td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td></td>
</tr>
<tr>
<td><strong>PMAXSW—Maximum of Packed Signed Word Integers</strong></td>
<td>01100110:00001111:11101110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td><strong>PMAXUB—Maximum of Packed Unsigned Byte Integers</strong></td>
<td>01100110:00001111:11011110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td><strong>PMINSW—Minimum of Packed Signed Word Integers</strong></td>
<td>01100110:00001111:11101010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
<tr>
<td><strong>PMINUB—Minimum of Packed Unsigned Byte Integers</strong></td>
<td>01100110:00001111:11011010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td></td>
</tr>
</tbody>
</table>
INSTRUCTION FORMATS AND ENCODINGS

Table B-21. Formats and Encodings of SSE2 Integer Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMOVMSKB - Move Byte Mask To Integer</td>
<td>0000110:111010111:11101011:11 r32 xmmreg</td>
</tr>
<tr>
<td>PMULHUW - Packed multiplication, store high word (unsigned)</td>
<td>0110 0110:0000 1111:1110 0100: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>PMULHW - Packed Multiplication, store high word</td>
<td>01100110:0000 1111:11100101: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>PMULLW - Packed Multiplication, store low word</td>
<td>01100110:0000 1111:11010101: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>PMULUDQ—Multiply Packed Unsigned Doubleword Integers</td>
<td>01100110:00001111:11110100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>POR - Bitwise Or</td>
<td>01100110:0000 1111:11101011: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>PSADDBW—Compute Sum of Absolute Differences</td>
<td>01100110:00001111:11110100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>PSHUFLW—Shuffle Packed Low Words</td>
<td>01100110:00001111:11110100:11 mod xmmreg r/m</td>
</tr>
<tr>
<td>PSHUFW—Shuffle Packed High Words</td>
<td>01100110:00001111:11110100:11 mod xmmreg r/m</td>
</tr>
</tbody>
</table>

Vol. 2B  B-43
### Table B-21. Formats and Encodings of SSE2 Integer Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PSHUFD—Shuffle Packed Doublewords</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg, imm8</td>
<td>01100110:00001111:01110000:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>01100110:00001111:01110000:11 mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PSLLDQ—Shift Double Quadword Left Logical</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg, imm8</td>
<td>01100110:00001111:01110011:11 111 xmmreg: imm8</td>
</tr>
<tr>
<td><strong>PSLL—Packed Shift Left Logical</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg1 by xmmreg2</td>
<td>01100110:0000 1111:1111100gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg by memory</td>
<td>01100110:0000 1111:1111100gg: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg by immediate</td>
<td>01100110:0000 1111:011100gg: 11 110 xmmreg: imm8 data</td>
</tr>
<tr>
<td><strong>PSRA—Packed Shift Right Arithmetic</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg1 by xmmreg2</td>
<td>01100110:0000 1111:111000gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg by memory</td>
<td>01100110:0000 1111:111000gg: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg by immediate</td>
<td>01100110:0000 1111:011100gg: 11 100 xmmreg: imm8 data</td>
</tr>
<tr>
<td><strong>PSRLDQ—Shift Double Quadword Right Logical</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg, imm8</td>
<td>01100110:00001111:01110011:11 011 xmmreg: imm8</td>
</tr>
<tr>
<td><strong>PSRL—Packed Shift Right Logical</strong></td>
<td></td>
</tr>
<tr>
<td>xmmxreg1 by xmmxreg2</td>
<td>01100110:0000 1111:110100gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmxreg by memory</td>
<td>01100110:0000 1111:110100gg: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmxreg by immediate</td>
<td>01100110:0000 1111:011100gg: 11 010 xmmreg: imm8 data</td>
</tr>
<tr>
<td><strong>PSUBQ—Subtract Packed Quadword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg to mmreg</td>
<td>00001111:11111011:11 xmmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>00001111:11111011: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:11111011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:11111011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PSUB—Subtract With Wrap-around</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 from xmmreg1</td>
<td>01100110:0000 1111:1111100gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory from xmmreg</td>
<td>01100110:0000 1111:1111100gg: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PSUBS—Subtract Signed With Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 from xmmreg1</td>
<td>01100110:0000 1111:1110100gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory from xmmreg</td>
<td>01100110:0000 1111:1110100gg: mod xmmreg r/m</td>
</tr>
</tbody>
</table>
### Table B-21. Formats and Encodings of SSE2 Integer Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSUBUS - Subtract Unsigned With Saturation</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 from xmmreg1</td>
<td>0000 1111:110110gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory from xmmreg</td>
<td>0000 1111:110110gg: mod xmmreg r/m</td>
</tr>
<tr>
<td>PUNPCKH—Unpack High Data To Next Larger Type</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:011010gg:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:011010gg: mod xmmreg r/m</td>
</tr>
<tr>
<td>PUNPCKHQDQ—Unpack High Data</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:011011101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:011011101: mod xmmreg r/m</td>
</tr>
<tr>
<td>PUNPCKL—Unpack Low Data To Next Larger Type</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:011000gg:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:011000gg: mod xmmreg r/m</td>
</tr>
<tr>
<td>PUNPCKLQDQ—Unpack Low Data</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:01101100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:01101100: mod xmmreg r/m</td>
</tr>
<tr>
<td>PXOR - Bitwise Xor</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>01100110:0000 1111:11101111: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>01100110:0000 1111:11101111: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

### Table B-22. Format and Encoding of SSE2 Cacheability Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASKMOVQ—Store Selected Bytes of Double Quadword</td>
<td></td>
</tr>
<tr>
<td>xmmreg to xmmreg</td>
<td>01100110:00001111:11101111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>CLFLUSH—Flush Cache Line</td>
<td></td>
</tr>
<tr>
<td>mem</td>
<td>00001111:11011010:mod r/m</td>
</tr>
<tr>
<td>MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint</td>
<td></td>
</tr>
<tr>
<td>xmmreg to mem</td>
<td>01100110:00001111:01101011: mod xmmreg r/m</td>
</tr>
<tr>
<td>MOVNTDQ—Store Double Quadword Using Non-Temporal Hint</td>
<td></td>
</tr>
<tr>
<td>xmmreg to mem</td>
<td>01100110:00001111:11100111: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

---

Vol. 2B  B-45
### 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>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVNTI—Store Doubleword Using Non-Temporal Hint reg to mem</td>
<td>00001111:11000011: mod reg r/m</td>
</tr>
<tr>
<td>PAUSE—Spin Loop Hint</td>
<td>11110011:10010000</td>
</tr>
<tr>
<td>LFENCE—Load Fence</td>
<td>00001111:10101110: 11 101 000</td>
</tr>
<tr>
<td>MFENCE—Memory Fence</td>
<td>00001111:10101110: 11 110 000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDSUBPD—Add /Sub packed DP FP numbers from XMM2/Mem to XMM1 xmmreg2 to xmmreg1</td>
<td>01100110:00001111:11010000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:11010000: mod xmmreg r/m</td>
</tr>
<tr>
<td>ADDSUBPS—Add /Sub packed SP FP numbers from XMM2/Mem to XMM1 xmmreg2 to xmmreg1</td>
<td>11110010:00001111:11010000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110010:00001111:11010000: mod xmmreg r/m</td>
</tr>
<tr>
<td>HADDPD—Add horizontally packed DP FP numbers XMM2/Mem to XMM1 xmmreg2 to xmmreg1</td>
<td>01100110:00001111:01111100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:01111100: mod xmmreg r/m</td>
</tr>
<tr>
<td>HADDPD—Add horizontally packed SP FP numbers XMM2/Mem to XMM1 xmmreg2 to xmmreg1</td>
<td>11110010:00001111:01111100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110010:00001111:01111100: mod xmmreg r/m</td>
</tr>
<tr>
<td>HSUBPD—Sub horizontally packed DP FP numbers XMM2/Mem to XMM1 xmmreg2 to xmmreg1</td>
<td>01100110:00001111:01111101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:01111101: mod xmmreg r/m</td>
</tr>
</tbody>
</table>
### Table B-23. Formats and Encodings of SSE3 Floating-Point Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSUBPS — Sub horizontally packed SP FP numbers XMM2/Mem to XMM1</td>
<td>11110010:00001111:01111101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>11110010:00001111:01111101: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110010:00001111:01111101: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

### Table B-24. Formats and Encodings for SSE3 Event Management Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONITOR — Set up a linear address range to be monitored by hardware</td>
<td>0000 1111 : 0000 0001:11 001 000</td>
</tr>
<tr>
<td>eax, ecx, edx</td>
<td>0000 1111 : 0000 0001:11 001 001</td>
</tr>
<tr>
<td>MWAIT — Wait until write-back store performed within the range specified by the instruction MONITOR</td>
<td>0000 1111 : 0000 0001:11 001 001</td>
</tr>
<tr>
<td>eax, ecx</td>
<td>0000 1111 : 0000 0001:11 001 001</td>
</tr>
</tbody>
</table>

### Table B-25. Formats and Encodings for SSE3 Integer and Move Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>FISTTP — Store ST in int16 (chop) and pop m16int</td>
<td>11011 111 : modA 001 r/m</td>
</tr>
<tr>
<td>FISTTP — Store ST in int32 (chop) and pop m32int</td>
<td>11011 011 : modA 001 r/m</td>
</tr>
<tr>
<td>FISTTP — Store ST in int64 (chop) and pop m64int</td>
<td>11011 101 : modA 001 r/m</td>
</tr>
<tr>
<td>LDDQU — Load unaligned integer 128-bit xmm, m128</td>
<td>11110010:00001111:11110000: modA xmmreg r/m</td>
</tr>
<tr>
<td>MOVVDDUP — Move 64 bits representing one DP data from XMM2/Mem to XMM1 and duplicate</td>
<td>11110010:00001111:00010010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>11110010:00001111:00010010: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110010:00001111:00010010: mod xmmreg r/m</td>
</tr>
<tr>
<td>MOVSHDUP — Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate high</td>
<td>11110011:00001111:00010110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>11110011:00001111:00010110: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110011:00001111:00010110: mod xmmreg r/m</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVSLDUP — Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate low</td>
<td>11110011:00001111:00010010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110011:00001111:00010010: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

Table B-25. Formats and Encodings for SSE3 Integer and Move Instructions (Contd.)
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>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2XM1 – Compute $2^{ST(0)} - 1$</td>
<td>11011 001 : 1111 0000</td>
</tr>
<tr>
<td>FABS – Absolute Value</td>
<td>11011 001 : 1110 0001</td>
</tr>
<tr>
<td>FADD – Add</td>
<td></td>
</tr>
<tr>
<td>ST(0) ← ST(0) + 32-bit memory</td>
<td>11011 000 : mod 000 r/m</td>
</tr>
<tr>
<td>ST(0) ← ST(0) + 64-bit memory</td>
<td>11011 100 : mod 000 r/m</td>
</tr>
<tr>
<td>ST(d) ← ST(0) + ST(i)</td>
<td>11011 d00 : 11 000 ST(i)</td>
</tr>
<tr>
<td>FADDP – Add and Pop</td>
<td></td>
</tr>
<tr>
<td>ST(0) ← ST(0) + ST(i)</td>
<td>11011 110 : 11 000 ST(i)</td>
</tr>
<tr>
<td>FBLD – Load Binary Coded Decimal</td>
<td>11011 111 : mod 100 r/m</td>
</tr>
<tr>
<td>FBSTP – Store Binary Coded Decimal and Pop</td>
<td>11011 111 : mod 110 r/m</td>
</tr>
<tr>
<td>FCHS – Change Sign</td>
<td>11011 001 : 1110 0000</td>
</tr>
<tr>
<td>FCLEX – Clear Exceptions</td>
<td>11011 011 : 1110 0010</td>
</tr>
<tr>
<td>FCOM – Compare Real</td>
<td></td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 000 : mod 010 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 100 : mod 010 r/m</td>
</tr>
<tr>
<td>ST(i)</td>
<td>11011 000 : 11 010 ST(i)</td>
</tr>
<tr>
<td>FCOMP – Compare Real and Pop</td>
<td></td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 000 : mod 011 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 100 : mod 011 r/m</td>
</tr>
<tr>
<td>ST(i)</td>
<td>11011 000 : 11 011 ST(i)</td>
</tr>
<tr>
<td>FCOMPP – Compare Real and Pop Twice</td>
<td>11011 110 : 11 011 001</td>
</tr>
<tr>
<td>FCOMIP – Compare Real, Set EFLAGS, and Pop</td>
<td>11011 111 : 11 110 ST(i)</td>
</tr>
<tr>
<td>FCOS – Cosine of ST(0)</td>
<td>11011 001 : 1111 1111</td>
</tr>
<tr>
<td>FDECSTP – Decrement Stack-Top Pointer</td>
<td>11011 001 : 1111 0110</td>
</tr>
<tr>
<td>FDIV – Divide</td>
<td></td>
</tr>
<tr>
<td>ST(0) ← ST(0) + 32-bit memory</td>
<td>11011 000 : mod 110 r/m</td>
</tr>
<tr>
<td>ST(0) ← ST(0) + 64-bit memory</td>
<td>11011 100 : mod 110 r/m</td>
</tr>
<tr>
<td>ST(d) ← ST(0) + ST(i)</td>
<td>11011 d00 : 1111 R ST(i)</td>
</tr>
</tbody>
</table>
## Table B-27. Floating-Point Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FDIVP – Divide and Pop</strong></td>
<td></td>
</tr>
<tr>
<td>ST(0) ← ST(0) + ST(i)</td>
<td>11011 110 : 1111 1 ST(i)</td>
</tr>
<tr>
<td><strong>FDIVR – Reverse Divide</strong></td>
<td></td>
</tr>
<tr>
<td>ST(0) ← 32-bit memory + ST(0)</td>
<td>11011 000 : mod 111 r/m</td>
</tr>
<tr>
<td>ST(0) ← 64-bit memory + ST(0)</td>
<td>11011 100 : mod 111 r/m</td>
</tr>
<tr>
<td>ST(d) ← ST(i) + ST(0)</td>
<td>11011 d00 : 1111 R ST(i)</td>
</tr>
<tr>
<td><strong>FDIVRP – Reverse Divide and Pop</strong></td>
<td></td>
</tr>
<tr>
<td>ST(0) ← ST(i) + ST(0)</td>
<td>11011 110 : 1111 0 ST(i)</td>
</tr>
<tr>
<td><strong>FFREE – Free ST(i) Register</strong></td>
<td>11011 101 : 1100 0 ST(i)</td>
</tr>
<tr>
<td><strong>FIADD – Add Integer</strong></td>
<td></td>
</tr>
<tr>
<td>ST(0) ← ST(0) + 16-bit memory</td>
<td>11011 110 : mod 000 r/m</td>
</tr>
<tr>
<td>ST(0) ← ST(0) + 32-bit memory</td>
<td>11011 010 : mod 000 r/m</td>
</tr>
<tr>
<td><strong>FICOM – Compare Integer</strong></td>
<td></td>
</tr>
<tr>
<td>16-bit memory</td>
<td>11011 110 : mod 010 r/m</td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 010 : mod 010 r/m</td>
</tr>
<tr>
<td><strong>FICOMP – Compare Integer and Pop</strong></td>
<td></td>
</tr>
<tr>
<td>16-bit memory</td>
<td>11011 110 : mod 011 r/m</td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 010 : mod 011 r/m</td>
</tr>
<tr>
<td><strong>FIDIV</strong></td>
<td></td>
</tr>
<tr>
<td>ST(0) ← ST(0) + 16-bit memory</td>
<td>11011 110 : mod 110 r/m</td>
</tr>
<tr>
<td>ST(0) ← ST(0) + 32-bit memory</td>
<td>11011 010 : mod 110 r/m</td>
</tr>
<tr>
<td><strong>FIDIVR</strong></td>
<td></td>
</tr>
<tr>
<td>ST(0) ← 16-bit memory ÷ ST(0)</td>
<td>11011 110 : mod 111 r/m</td>
</tr>
<tr>
<td>ST(0) ← 32-bit memory ÷ ST(0)</td>
<td>11011 010 : mod 111 r/m</td>
</tr>
<tr>
<td><strong>FILD – Load Integer</strong></td>
<td></td>
</tr>
<tr>
<td>16-bit memory</td>
<td>11011 111 : mod 000 r/m</td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 011 : mod 000 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 111 : mod 101 r/m</td>
</tr>
<tr>
<td><strong>FIMUL</strong></td>
<td></td>
</tr>
<tr>
<td>ST(0) ← ST(0) × 16-bit memory</td>
<td>11011 110 : mod 001 r/m</td>
</tr>
<tr>
<td>ST(0) ← ST(0) × 32-bit memory</td>
<td>11011 010 : mod 001 r/m</td>
</tr>
<tr>
<td><strong>FINCSTP – Increment Stack Pointer</strong></td>
<td>11011 001 : 1111 0111</td>
</tr>
<tr>
<td><strong>FINIT – Initialize Floating-Point Unit</strong></td>
<td></td>
</tr>
</tbody>
</table>
### Table B-27. Floating-Point Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FIST – Store Integer</strong></td>
<td>11011 111 : mod 010 r/m</td>
</tr>
<tr>
<td>16-bit memory</td>
<td>11011 011 : mod 010 r/m</td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 111 : mod 011 r/m</td>
</tr>
<tr>
<td><strong>FISTP – Store Integer and Pop</strong></td>
<td>11011 011 : mod 011 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 111 : mod 111 r/m</td>
</tr>
<tr>
<td><strong>FISUB</strong></td>
<td>11011 110 : mod 100 r/m</td>
</tr>
<tr>
<td>ST(0) ← ST(0) - 16-bit memory</td>
<td>11011 010 : mod 100 r/m</td>
</tr>
<tr>
<td>ST(0) ← ST(0) - 32-bit memory</td>
<td>11011 011 : mod 101 r/m</td>
</tr>
<tr>
<td><strong>FISUBR</strong></td>
<td>11011 010 : mod 101 r/m</td>
</tr>
<tr>
<td>ST(0) ← 16-bit memory − ST(0)</td>
<td>11011 010 : mod 101 r/m</td>
</tr>
<tr>
<td>ST(0) ← 32-bit memory − ST(0)</td>
<td>11011 011 : mod 101 r/m</td>
</tr>
<tr>
<td><strong>FLD – Load Real</strong></td>
<td>11011 001 : mod 000 r/m</td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 010 : mod 001 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 011 : mod 001 r/m</td>
</tr>
<tr>
<td>80-bit memory</td>
<td>11011 101 : mod 001 r/m</td>
</tr>
<tr>
<td>ST(i)</td>
<td>11011 001 : mod 001 r/m</td>
</tr>
<tr>
<td><strong>FLD1 – Load +1.0 into ST(0)</strong></td>
<td>11011 001 : mod 100 1000</td>
</tr>
<tr>
<td><strong>FLDCW – Load Control Word</strong></td>
<td>11011 001 : mod 101 r/m</td>
</tr>
<tr>
<td><strong>FLDENV – Load FPU Environment</strong></td>
<td>11011 001 : mod 100 r/m</td>
</tr>
<tr>
<td><strong>FLDL2E – Load log₂(ε) into ST(0)</strong></td>
<td>11011 001 : 11101010</td>
</tr>
<tr>
<td><strong>FLDL2T – Load log₂(10) into ST(0)</strong></td>
<td>11011 001 : 11101001</td>
</tr>
<tr>
<td><strong>FLDLG2 – Load log₁₀(2) into ST(0)</strong></td>
<td>11011 001 : 11101100</td>
</tr>
<tr>
<td><strong>FLDLN2 – Load log₂(2) into ST(0)</strong></td>
<td>11011 001 : 11101101</td>
</tr>
<tr>
<td><strong>FLDPI – Load π into ST(0)</strong></td>
<td>11011 001 : 11101011</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>FLDZ – Load +0.0 into ST(0)</td>
<td>11011 001 : 1110 1110</td>
</tr>
<tr>
<td>FMUL – Multiply</td>
<td></td>
</tr>
<tr>
<td>ST(0) ← ST(0) × 32-bit memory</td>
<td>11011 000 : mod 001 r/m</td>
</tr>
<tr>
<td>ST(0) ← ST(0) × 64-bit memory</td>
<td>11011 100 : mod 001 r/m</td>
</tr>
<tr>
<td>ST(d) ← ST(0) × ST(i)</td>
<td>11011 d00 : 1100 1 ST(i)</td>
</tr>
<tr>
<td>FMULP – Multiply</td>
<td></td>
</tr>
<tr>
<td>ST(i) ← ST(0) × ST(i)</td>
<td>11011 110 : 1100 1 ST(i)</td>
</tr>
<tr>
<td>FNOP – No Operation</td>
<td>11011 001 : 1101 0000</td>
</tr>
<tr>
<td>FPATAN – Partial Arctangent</td>
<td>11011 001 : 1111 0011</td>
</tr>
<tr>
<td>FPREM – Partial Remainder</td>
<td>11011 001 : 1111 1000</td>
</tr>
<tr>
<td>FPREM1 – Partial Remainder (IEEE)</td>
<td>11011 001 : 1111 0101</td>
</tr>
<tr>
<td>FPTAN – Partial Tangent</td>
<td>11011 001 : 1111 0010</td>
</tr>
<tr>
<td>FRNDINT – Round to Integer</td>
<td>11011 001 : 1111 1100</td>
</tr>
<tr>
<td>FRSTOR – Restore FPU State</td>
<td>11011 101 : mod 100 r/m</td>
</tr>
<tr>
<td>FSAVE – Store FPU State</td>
<td>11011 101 : mod 110 r/m</td>
</tr>
<tr>
<td>FSCALE – Scale</td>
<td>11011 001 : 1111 1101</td>
</tr>
<tr>
<td>FSIN – Sine</td>
<td>11011 001 : 1111 1110</td>
</tr>
<tr>
<td>FSINCOS – Sine and Cosine</td>
<td>11011 001 : 1111 1011</td>
</tr>
<tr>
<td>FSQRT – Square Root</td>
<td>11011 001 : 1111 1010</td>
</tr>
<tr>
<td>FST – Store Real</td>
<td></td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 001 : mod 010 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 101 : mod 010 r/m</td>
</tr>
<tr>
<td>ST(i)</td>
<td>11011 101 : 11 010 ST(i)</td>
</tr>
<tr>
<td>FSTCW – Store Control Word</td>
<td>11011 001 : mod 111 r/m</td>
</tr>
<tr>
<td>FSTENV – Store FPU Environment</td>
<td>11011 001 : mod 110 r/m</td>
</tr>
<tr>
<td>FSTP – Store Real and Pop</td>
<td></td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 001 : mod 011 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 101 : mod 011 r/m</td>
</tr>
<tr>
<td>80-bit memory</td>
<td>11011 011 : mod 111 r/m</td>
</tr>
<tr>
<td>ST(i)</td>
<td>11011 101 : 11 011 ST(i)</td>
</tr>
<tr>
<td>FSTS – Store Status Word into AX</td>
<td>11011 111 : 1110 0000</td>
</tr>
<tr>
<td>FSTSW – Store Status Word into Memory</td>
<td>11011 101 : mod 111 r/m</td>
</tr>
</tbody>
</table>
Table B-27. Floating-Point Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FSUB – Subtract</strong></td>
<td></td>
</tr>
<tr>
<td>ST(0) ← ST(0) – 32-bit memory</td>
<td>11011 000 : mod 100 r/m</td>
</tr>
<tr>
<td>ST(0) ← ST(0) – 64-bit memory</td>
<td>11011 100 : mod 100 r/m</td>
</tr>
<tr>
<td>ST(d) ← ST(0) – ST(i)</td>
<td>11011 d00 : 1110 R ST(i)</td>
</tr>
<tr>
<td><strong>FSUBP – Subtract and Pop</strong></td>
<td></td>
</tr>
<tr>
<td>ST(0) ← ST(0) – ST(i)</td>
<td>11011 110 : 1110 ST(i)</td>
</tr>
<tr>
<td><strong>FSUBR – Reverse Subtract</strong></td>
<td></td>
</tr>
<tr>
<td>ST(0) ← 32-bit memory – ST(0)</td>
<td>11011 000 : mod 101 r/m</td>
</tr>
<tr>
<td>ST(0) ← 64-bit memory – ST(0)</td>
<td>11011 100 : mod 101 r/m</td>
</tr>
<tr>
<td>ST(d) ← ST(i) – ST(0)</td>
<td>11011 d00 : 1110 R ST(i)</td>
</tr>
<tr>
<td><strong>FSUBRP – Reverse Subtract and Pop</strong></td>
<td></td>
</tr>
<tr>
<td>ST(i) ← ST(i) – ST(0)</td>
<td>11011 110 : 1110 ST(i)</td>
</tr>
<tr>
<td><strong>FTST – Test</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 001 : 1110 0100</td>
</tr>
<tr>
<td><strong>FUCOM – Unordered Compare Real</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 101 : 1110 ST(i)</td>
</tr>
<tr>
<td><strong>FUCOMP – Unordered Compare Real and Pop</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 101 : 1110 1 ST(i)</td>
</tr>
<tr>
<td><strong>FUCOMPP – Unordered Compare Real and Pop Twice</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 010 : 1110 1001</td>
</tr>
<tr>
<td><strong>FUCOMI – Unordered Compare Real and Set EFLAGS</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 011 : 11 101 ST(i)</td>
</tr>
<tr>
<td><strong>FUCOMIP – Unordered Compare Real, Set EFLAGS, and Pop</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 111 : 11 101 ST(i)</td>
</tr>
<tr>
<td><strong>FXAM – Examine</strong></td>
<td>11011 001 : 1110 0101</td>
</tr>
<tr>
<td><strong>FXCH – Exchange ST(0) and ST(i)</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 001 : 1100 1 ST(i)</td>
</tr>
<tr>
<td><strong>FXTRACT – Extract Exponent and Significand</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11011 001 : 1111 0100</td>
</tr>
<tr>
<td><strong>FYL2X – ST(1) × log₂(ST(0))</strong></td>
<td>11011 001 : 1111 0001</td>
</tr>
<tr>
<td><strong>FYL2XP1 – ST(1) × log₂(ST(0) + 1.0)</strong></td>
<td>11011 001 : 1111 1001</td>
</tr>
<tr>
<td><strong>FWAIT – Wait until FPU Ready</strong></td>
<td>1001 1011</td>
</tr>
</tbody>
</table>
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
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:

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
</tr>
<tr>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td>2</td>
<td>127</td>
</tr>
</tbody>
</table>
```

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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPD</td>
<td>_m128d_mm_add_pd(_m128d a, _m128d b)</td>
<td>Adds the two DP FP (double-precision, floating-point) values of a and b.</td>
</tr>
<tr>
<td>ADDPS</td>
<td>_m128_mm_add_ps(_m128 a, _m128 b)</td>
<td>Adds the four SP FP (single-precision, floating-point) values of a and b.</td>
</tr>
<tr>
<td>ADDSD</td>
<td>_m128d_mm_add_sd(_m128d a, _m128d b)</td>
<td>Adds the lower DP FP values of a and b; the upper three DP FP values are passed through from a.</td>
</tr>
<tr>
<td>ADDSS</td>
<td>_m128_mm_add_ss(_m128 a, _m128 b)</td>
<td>Adds the lower SP FP values of a and b; the upper three SP FP values are passed through from a.</td>
</tr>
<tr>
<td>ADDSUBPD</td>
<td>_m128d_mm_addsub_pd(_m128d a, _m128d b)</td>
<td>Add/Subtract packed DP FP numbers from XMM2/Mem to XMM1.</td>
</tr>
<tr>
<td>ADDSUBPS</td>
<td>_m128_mm_addsub_ps(_m128 a, _m128 b)</td>
<td>Add/Subtract packed SP FP numbers from XMM2/Mem to XMM1.</td>
</tr>
<tr>
<td>ANDNPD</td>
<td>_m128d_mm_andnot_pd(_m128d a, _m128d b)</td>
<td>Computes the bitwise AND-NOT of the two DP FP values of a and b.</td>
</tr>
<tr>
<td>ANDNPS</td>
<td>_m128_mm_andnot_ps(_m128 a, _m128 b)</td>
<td>Computes the bitwise AND-NOT of the four SP FP values of a and b.</td>
</tr>
<tr>
<td>ANDPD</td>
<td>_m128d_mm_and_pd(_m128d a, _m128d b)</td>
<td>Computes the bitwise AND of the two DP FP values of a and b.</td>
</tr>
<tr>
<td>ANDPS</td>
<td>_m128_mm_and_ps(_m128 a, _m128 b)</td>
<td>Computes the bitwise AND of the four SP FP values of a and b.</td>
</tr>
<tr>
<td>CLFLUSH</td>
<td>void_mm_clflush(void const *p)</td>
<td>Cache line containing p is flushed and invalidated from all caches in the coherency domain.</td>
</tr>
<tr>
<td>CMPPD</td>
<td>_m128d_mm_cmpeq_pd(_m128d a, _m128d b)</td>
<td>Compare for equality.</td>
</tr>
<tr>
<td></td>
<td>_m128d_mm_cmplt_pd(_m128d a, _m128d b)</td>
<td>Compare for less-than.</td>
</tr>
<tr>
<td></td>
<td>_m128d_mm_cmple_pd(_m128d a, _m128d b)</td>
<td>Compare for less-than-or-equal.</td>
</tr>
<tr>
<td></td>
<td>_m128d_mm_cmpge_pd(_m128d a, _m128d b)</td>
<td>Compare for greater-than.</td>
</tr>
<tr>
<td></td>
<td>_m128d_mm_cmpgt_pd(_m128d a, _m128d b)</td>
<td></td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>__m128d _mm_cmpeq_pd(__m128d a, __m128d b)</td>
<td>Compare for equality.</td>
<td></td>
</tr>
<tr>
<td>__m128d _mm_cmpneq_pd(__m128d a, __m128d b)</td>
<td>Compare for inequality.</td>
<td></td>
</tr>
<tr>
<td>__m128d _mm_cmpnlt_pd(__m128d a, __m128d b)</td>
<td>Compare for not-less-than.</td>
<td></td>
</tr>
<tr>
<td>__m128d _mm_cmpnge_pd(__m128d a, __m128d b)</td>
<td>Compare for not-greater-than-or-equal.</td>
<td></td>
</tr>
<tr>
<td>__m128d _mm_cmpord_pd(__m128d a, __m128d b)</td>
<td>Compare for ordered.</td>
<td></td>
</tr>
<tr>
<td>__m128d _mm_cmpunord_pd(__m128d a, __m128d b)</td>
<td>Compare for unordered.</td>
<td></td>
</tr>
<tr>
<td>__m128d _mm_cmpnle_pd(__m128d a, __m128d b)</td>
<td>Compare for not-less-than-or-equal.</td>
<td></td>
</tr>
</tbody>
</table>

**CMPPS**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__m128 _mm_cmpeq_ps(__m128 a, __m128 b)</td>
<td>Compare for equality.</td>
<td></td>
</tr>
<tr>
<td>__m128 _mm_cmpneq_ps(__m128 a, __m128 b)</td>
<td>Compare for inequality.</td>
<td></td>
</tr>
<tr>
<td>__m128 _mm_cmpnlt_ps(__m128 a, __m128 b)</td>
<td>Compare for not-less-than.</td>
<td></td>
</tr>
<tr>
<td>__m128 _mm_cmpnge_ps(__m128 a, __m128 b)</td>
<td>Compare for not-greater-than-or-equal.</td>
<td></td>
</tr>
<tr>
<td>__m128 _mm_cmpord_ps(__m128 a, __m128 b)</td>
<td>Compare for ordered.</td>
<td></td>
</tr>
<tr>
<td>__m128 _mm_cmpunord_ps(__m128 a, __m128 b)</td>
<td>Compare for unordered.</td>
<td></td>
</tr>
</tbody>
</table>

**CMPSD**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__m128d _mm_cmpeq_sd(__m128d a, __m128d b)</td>
<td>Compare for equality.</td>
<td></td>
</tr>
<tr>
<td>__m128d _mm_cmpneq_sd(__m128d a, __m128d b)</td>
<td>Compare for inequality.</td>
<td></td>
</tr>
<tr>
<td>__m128d _mm_cmpnlt_sd(__m128d a, __m128d b)</td>
<td>Compare for not-less-than.</td>
<td></td>
</tr>
<tr>
<td>__m128d _mm_cmpnge_sd(__m128d a, __m128d b)</td>
<td>Compare for not-greater-than-or-equal.</td>
<td></td>
</tr>
<tr>
<td>__m128d _mm_cmpord_sd(__m128d a, __m128d b)</td>
<td>Compare for ordered.</td>
<td></td>
</tr>
<tr>
<td>__m128d _mm_cmpunord_sd(__m128d a, __m128d b)</td>
<td>Compare for unordered.</td>
<td></td>
</tr>
</tbody>
</table>
**Table C-1. Simple Intrinsics (Contd.)**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>_m128d_mm_cmpnle_sd(__m128d a, __m128d b)</td>
<td>Compare for not-greater-than.</td>
</tr>
<tr>
<td></td>
<td>_m128d_mm_cmpngt_sd(__m128d a, __m128d b)</td>
<td>Compare for not-greater-than-or-equal.</td>
</tr>
<tr>
<td></td>
<td>_m128d_mm_cmpnge_sd(__m128d a, __m128d b)</td>
<td>Compare for ordered.</td>
</tr>
<tr>
<td></td>
<td>_m128d_mm_cmpord_sd(__m128d a, __m128d b)</td>
<td>Compare for unordered.</td>
</tr>
<tr>
<td></td>
<td>_m128d_mm_cmpunord_sd(__m128d a, __m128d b)</td>
<td>Compare for not-less-than-or-equal.</td>
</tr>
<tr>
<td>CMPSS</td>
<td>_m128_mm_cmpeq_ss(__m128 a, __m128 b)</td>
<td>Compare for equality.</td>
</tr>
<tr>
<td></td>
<td>_m128_mm_cmplt_ss(__m128 a, __m128 b)</td>
<td>Compare for less-than.</td>
</tr>
<tr>
<td></td>
<td>_m128_mm_cmple_ss(__m128 a, __m128 b)</td>
<td>Compare for less-than-or-equal.</td>
</tr>
<tr>
<td></td>
<td>_m128_mm_cmpgt_ss(__m128 a, __m128 b)</td>
<td>Compare for greater-than.</td>
</tr>
<tr>
<td></td>
<td>_m128_mm_CMPGE_S_S(__m128 a, __m128 b)</td>
<td>Compare for greater-than-or-equal.</td>
</tr>
<tr>
<td></td>
<td>_m128_mm_cmpeq_ss(__m128 a, __m128 b)</td>
<td>Compare for inequality.</td>
</tr>
<tr>
<td></td>
<td>_m128_mm_cmplt_ss(__m128 a, __m128 b)</td>
<td>Compare for not-less-than.</td>
</tr>
<tr>
<td></td>
<td>_m128_mm_cmple_ss(__m128 a, __m128 b)</td>
<td>Compare for not-greater-than.</td>
</tr>
<tr>
<td></td>
<td>_m128_mm_cmpord_sd(__m128d a, __m128d b)</td>
<td>Compare for ordered.</td>
</tr>
<tr>
<td></td>
<td>_m128d_mm_cmpunord_sd(__m128d a, __m128d b)</td>
<td>Compare for not-less-than-or-equal.</td>
</tr>
<tr>
<td>COMISD</td>
<td>int _mm_comieq_sd(__m128d a, __m128d b)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>int _mm_comilt_sd(__m128d a, __m128d b)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>int _mm_comile_sd(__m128d a, __m128d b)</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>int _mm_comigt_sd(__m128d a, __m128d b)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>int _mm_comige_sd(__m128d a, __m128d b)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>int _mm_comineq_sd(__m128d a, __m128d b)</td>
<td>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.</td>
</tr>
<tr>
<td>COMISS</td>
<td>int _mm_comieq_ss(__m128 a, __m128 b)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>int _mm_comilt_ss(__m128 a, __m128 b)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>int _mm_comile_ss(__m128 a, __m128 b)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>int _mm_comigt_ss(__m128 a, __m128 b)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>int _mm_comige_ss(__m128 a, __m128 b)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>int _mm_comineq_ss(__m128 a, __m128 b)</td>
<td>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.</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>CVTDQ2PD</td>
<td>_m128d_mm_cvtepi32_pd(__m128i a)</td>
<td>Convert the lower two 32-bit signed integer values in packed form in a to two DP FP values.</td>
</tr>
<tr>
<td>CVTDQ2PS</td>
<td>_m128_mm_cvtepi32_ps(__m128i a)</td>
<td>Convert the four 32-bit signed integer values in packed form in a to four SP FP values.</td>
</tr>
<tr>
<td>CVTPD2DQ</td>
<td>_m128i_mm_cvtpd_epi32(__m128d a)</td>
<td>Convert the two DP FP values in a to two 32-bit signed integer values.</td>
</tr>
<tr>
<td>CVTPD2PI</td>
<td>_m64_mm_cvtpd_pi32(__m128d a)</td>
<td>Convert the two DP FP values in a to two 32-bit signed integer values.</td>
</tr>
<tr>
<td>CVTPD2PS</td>
<td>_m128_mm_cvtpd_ps(__m128d a)</td>
<td>Convert the two DP FP values in a to two SP FP values.</td>
</tr>
<tr>
<td>CVTPI2PD</td>
<td>_m128d_mm_cvtpi32_pd(__m64 a)</td>
<td>Convert the two 32-bit integer values in a to two DP FP values.</td>
</tr>
<tr>
<td>CVTPI2PS</td>
<td>_m128_mm_cvtpi32_ps(__m128 a, __m64 b)</td>
<td>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.</td>
</tr>
<tr>
<td>CVTPS2DQ</td>
<td>_m128i_mm_cvtps_epi32(__m128 a)</td>
<td>Convert four SP FP values in a to four 32-bit signed integers according to the current rounding mode.</td>
</tr>
<tr>
<td>CVTPS2PD</td>
<td>_m128d_mm_cvtps_pd(__m128 a)</td>
<td>Convert the lower two SP FP values in a to DP FP values.</td>
</tr>
<tr>
<td>CVTPS2PI</td>
<td>_m64_mm_cvtpi32_ps(__m128 a)</td>
<td>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.</td>
</tr>
<tr>
<td>CVTSD2SI</td>
<td>int_mm_cvtsd_si32(__m128d a)</td>
<td>Convert the lower DP FP value in a to a 32-bit integer value.</td>
</tr>
<tr>
<td>CVTSD2SS</td>
<td>_m128_mm_cvtsd_ss(__m128 a, __m128d b)</td>
<td>Convert the lower DP FP value in b to a SP FP value; the upper three SP FP values of a are passed through.</td>
</tr>
<tr>
<td>CVTSI2SD</td>
<td>_m128d_mm_cvtisi32_sd(__m128d a, int b)</td>
<td>Convert the 32-bit integer value b to a DP FP value; the upper DP FP values are passed through from a.</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CVTSI2SS</td>
<td>__m128_mm_cvtsi2ss(__m128 a, int b)</td>
<td>Convert the 32-bit integer value b to an SP FP value; the upper three SP FP values are passed through from a.</td>
</tr>
<tr>
<td>CVTSS2SD</td>
<td>__m128d__mm_cvtsd(__m128d a, __m128 d b)</td>
<td>Convert the lower SP FP value of b to DP FP value, the upper DP FP value is passed through from a.</td>
</tr>
<tr>
<td>CVTSS2SI</td>
<td>int_mm_cvtsi32_sisi32(__m128 a)</td>
<td>Convert the lower SP FP value of a to a 32-bit integer.</td>
</tr>
<tr>
<td>CVTTPD2DQ</td>
<td>__m128i__mm_cvttpd_epi32(__m128d a)</td>
<td>Convert the two DP FP values of a to two 32-bit signed integer values with truncation, the upper two integer values are 0.</td>
</tr>
<tr>
<td>CVTTPD2PI</td>
<td>__m64__mm_cvttpd_pi32(__m128d a)</td>
<td>Convert the two DP FP values of a to 32-bit signed integer values with truncation.</td>
</tr>
<tr>
<td>CVTTPS2DQ</td>
<td>__m128i__mm_cvttps_epi32(__m128 a)</td>
<td>Convert four SP FP values of a to four 32-bit integer with truncation.</td>
</tr>
<tr>
<td>CVTTPS2PI</td>
<td>__m64__mm_cvtt_ps2pi(_m128 a) __m64__mm_cvtt_ps2pi(_m128 a)</td>
<td>Convert the two lower SP FP values of a to two 32-bit integer with truncation, returning the integers in packed form.</td>
</tr>
<tr>
<td>CVTTS2SI</td>
<td>int_mm_cvttss_sisi32(__m128 a)</td>
<td>Convert the lower DP FP value of a to a 32-bit signed integer using truncation.</td>
</tr>
<tr>
<td>CVTTSS2SI</td>
<td>int_mm_cvtt_ss2ssi(__m128 a) int_mm_cvttss_sisi32(__m128 a)</td>
<td>Convert the lower SP FP value of a to a 32-bit integer according to the current rounding mode.</td>
</tr>
<tr>
<td></td>
<td>__m64__mm_cvtsi32_sisi64(int i)</td>
<td>Convert the integer object i to a 64-bit __m64 object. The integer value is zero extended to 64 bits.</td>
</tr>
<tr>
<td></td>
<td>int_mm_cvtsi64_sisi32(__m128 a) __m64__mm_cvttss_sisi64(int i)</td>
<td>Convert the lower 32 bits of the __m64 object m to an integer.</td>
</tr>
<tr>
<td>DIVPD</td>
<td>__m128d__mm_div_pd(__m128d a, __m128d b)</td>
<td>Divides the two DP FP values of a and b.</td>
</tr>
<tr>
<td>DIVPS</td>
<td>__m128__mm_div_ps(__m128 a, __m128 b)</td>
<td>Divides the four SP FP values of a and b.</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>DIVSD</td>
<td>__m128d _mm_div_sd(__m128d a, __m128d b)</td>
<td>Divides the lower DP FP values of a and b; the upper three DP FP values are passed through from a.</td>
</tr>
<tr>
<td>DIVSS</td>
<td>__m128 _mm_div_ss(__m128 a, __m128 b)</td>
<td>Divides the lower SP FP values of a and b; the upper three SP FP values are passed through from a.</td>
</tr>
<tr>
<td>EMMS</td>
<td>void _mm_empty()</td>
<td>Clears the MMX technology state.</td>
</tr>
<tr>
<td>HADDPD</td>
<td>__m128d _mm_hadd_pd(__m128d a, __m128d b)</td>
<td>Add horizontally packed DP FP numbers from XMM2/Mem to XMM1.</td>
</tr>
<tr>
<td>HADDPD</td>
<td>__m128 _mm_hadd_ps(__m128 a, __m128 b)</td>
<td>Add horizontally packed SP FP numbers from XMM2/Mem to XMM1.</td>
</tr>
<tr>
<td>HSUBPD</td>
<td>__m128d _mm_hsub_pd(__m128d a, __m128d b)</td>
<td>Subtract horizontally packed DP FP numbers in XMM2/Mem from XMM1.</td>
</tr>
<tr>
<td>HSUBPS</td>
<td>__m128 _mm_hsub_ps(__m128 a, __m128 b)</td>
<td>Subtract horizontally packed SP FP numbers in XMM2/Mem from XMM1.</td>
</tr>
<tr>
<td>LDDQU</td>
<td>__m128i _mm_lddqu_si128(__m128i const *p)</td>
<td>Load 128 bits from Mem to XMM register.</td>
</tr>
<tr>
<td>LDMXCSR</td>
<td>_mm_setcsr(unsigned int i)</td>
<td>Sets the control register to the value specified.</td>
</tr>
<tr>
<td>LFENCE</td>
<td>void _mm_lfence(void)</td>
<td>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.</td>
</tr>
<tr>
<td>MASKMOVDQU</td>
<td>void _mm_maskmoveu_si128(__m128i d, __m128i n, char *p)</td>
<td>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.</td>
</tr>
<tr>
<td>MASKMOVQ</td>
<td>void _mm_maskmove_si64(__m64 d, __m64 n, char *p)</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXPD</td>
<td>_m128d _mm_max_pd(__m128d a, __m128d b)</td>
<td>Computes the maximums of the two DP FP values of a and b.</td>
</tr>
<tr>
<td>MAXPS</td>
<td>_m128 _mm_max_ps(__m128 a, __m128 b)</td>
<td>Computes the maximums of the four SP FP values of a and b.</td>
</tr>
<tr>
<td>MAXSD</td>
<td>_m128d _mm_max_sd(__m128d a, __m128d b)</td>
<td>Computes the maximum of the lower DP FP values of a and b; the upper DP FP values are passed through from a.</td>
</tr>
<tr>
<td>MAXSS</td>
<td>_m128d _mm_max_ss(__m128d a, __m128d b)</td>
<td>Computes the maximum of the lower SP FP values of a and b; the upper three SP FP values are passed through from a.</td>
</tr>
<tr>
<td>MFENCE</td>
<td>void _mm_mfence(void)</td>
<td>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.</td>
</tr>
<tr>
<td>MINPD</td>
<td>_m128d _mm_min_pd(__m128d a, __m128d b)</td>
<td>Computes the minimums of the two DP FP values of a and b.</td>
</tr>
<tr>
<td>MINPS</td>
<td>_m128 _mm_min_ps(__m128 a, __m128 b)</td>
<td>Computes the minimums of the four SP FP values of a and b.</td>
</tr>
<tr>
<td>MINSD</td>
<td>_m128d _mm_min_sd(__m128d a, __m128d b)</td>
<td>Computes the minimum of the lower DP FP values of a and b; the upper DP FP values are passed through from a.</td>
</tr>
<tr>
<td>MINSS</td>
<td>_m128d _mm_min_ss(__m128d a, __m128d b)</td>
<td>Computes the minimum of the lower SP FP values of a and b; the upper three SP FP values are passed through from a.</td>
</tr>
<tr>
<td>MONITOR</td>
<td>void _mm_monitor(void const *p, unsigned extensions, unsigned hints)</td>
<td>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.</td>
</tr>
<tr>
<td>MOVAPD</td>
<td>_m128d _mm_load_pd(double * p)</td>
<td>Loads two DP FP values. The address p must be 16-byte-aligned.</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MOVAPS</td>
<td>__m128_mm_load_ps(float * p)</td>
<td>Loads four SP FP values. The address p must be 16-byte-aligned.</td>
</tr>
<tr>
<td></td>
<td>void_mm_store_ps(float *p, __m128 a)</td>
<td>Stores four SP FP values. The address p must be 16-byte-aligned.</td>
</tr>
<tr>
<td>MOVDDUP</td>
<td>__m128d_mm_movedup_pd(__m128d a)</td>
<td>Move 64 bits representing the lower DP data element from XMM2/Mem to XMM1 register and duplicate.</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_loaddup_pd(double const * dp)</td>
<td></td>
</tr>
<tr>
<td>MOVDQA</td>
<td>__m128i_mm_load_si128(__m128i * p)</td>
<td>Loads 128-bit values from p. The address p must be 16-byte-aligned.</td>
</tr>
<tr>
<td></td>
<td>void_mm_store_si128(__m128i *p, __m128i a)</td>
<td>Stores 128-bit value in a to address p. The address p must be 16-byte-aligned.</td>
</tr>
<tr>
<td>MOVDQU</td>
<td>__m128i_mm_loadu_si128(__m128i * p)</td>
<td>Loads 128-bit values from p. The address p need not be 16-byte-aligned.</td>
</tr>
<tr>
<td></td>
<td>void_mm_storeu_si128(__m128i *p, __m128i a)</td>
<td>Stores 128-bit value in a to address p. The address p need not be 16-byte-aligned.</td>
</tr>
<tr>
<td>MOVDQ2Q</td>
<td>__m64_mm_movepi64_pi64(__m128i a)</td>
<td>Return the lower 64-bits in a as __m64 type.</td>
</tr>
<tr>
<td>MOVHLPS</td>
<td>__m128_mm_movehl_ps(__m128 a, __m128 b)</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic Description</th>
</tr>
</thead>
</table>
| 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 lower 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. |
| MOVNNDQ   | 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.)**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVNTPD</td>
<td>void_mm_stream_pd(double * p, __m128d a)</td>
<td>Stores the data in a to the address p without polluting the caches. The address must be 16-byte-aligned.</td>
</tr>
<tr>
<td>MOVNTPS</td>
<td>void_mm_stream_ps(float * p, __m128 a)</td>
<td>Stores the data in a to the address p without polluting the caches. The address must be 16-byte-aligned.</td>
</tr>
<tr>
<td>MOVNTI</td>
<td>void_mm_stream_si32(int * p, int a)</td>
<td>Stores the data in a to the address p without polluting the caches.</td>
</tr>
<tr>
<td>MOVNTQ</td>
<td>void_mm_stream_pl(__m64 * p, __m64 a)</td>
<td>Stores the data in a to the address p without polluting the caches.</td>
</tr>
<tr>
<td>MOVQ</td>
<td>__m128i_mm_loadl_epi64(__m128i * p)</td>
<td>Loads the lower 64 bits from p into the lower 64 bits of destination and zero-extend the upper 64 bits.</td>
</tr>
<tr>
<td></td>
<td>void_mm_storel_epi64(__m128i * p, __m128i a)</td>
<td>Stores the lower 64 bits of a to the lower 64 bits at p.</td>
</tr>
<tr>
<td></td>
<td>__m128i_mm_move_epi64(__m128i a)</td>
<td>Moves the lower 64 bits of a to the lower 64 bits of destination. The upper 64 bits are cleared.</td>
</tr>
<tr>
<td>MOVQ2DQ</td>
<td>__m128i_mm_movpi64_epi64(__m64 a)</td>
<td>Move the 64 bits of a into the lower 64-bits, while zero-extending the upper bits.</td>
</tr>
<tr>
<td>MOVSD</td>
<td>__m128d_mm_load_sd(double * p)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>void_mm_store_sd(double * p, __m128d a)</td>
<td>Stores the lower DP FP value of a to address p. The address P need not be 16-byte aligned.</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_move_sd(__m128d a, __m128d b)</td>
<td>Sets the lower DP FP values of b to destination. The upper DP FP value is passed through from a.</td>
</tr>
<tr>
<td>MOVSHDUP</td>
<td>__m128_mm_movelup_ps(__m128 a)</td>
<td>Move 128 bits representing packed SP data elements from XMM2/Mem to XMM1 register and duplicate high.</td>
</tr>
</tbody>
</table>
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVSLDUP</td>
<td>__m128 _mm_moveldup_ps(__m128 a)</td>
<td>Move 128 bits representing packed SP data elements from XMM2/Mem to XMM1 register and duplicate low.</td>
</tr>
<tr>
<td>MOVSS</td>
<td>__m128 _mm_load_ss(float * p)</td>
<td>Loads an SP FP value into the low word and clears the upper three words.</td>
</tr>
<tr>
<td></td>
<td>void_mm_store_ss(float * p, __m128 a)</td>
<td>Stores the lower SP FP value.</td>
</tr>
<tr>
<td></td>
<td>__m128 _mm_move_ss(__m128 a, __m128 b)</td>
<td>Sets the low word to the SP FP value of b. The upper 3 SP FP values are passed through from a.</td>
</tr>
<tr>
<td>MOVUPD</td>
<td>__m128d _mm_loadu_pd(double * p)</td>
<td>Loads two DP FP values from p. The address p need not be 16-byte-aligned.</td>
</tr>
<tr>
<td></td>
<td>void_mm_storeu_pd(double *p, __m128d a)</td>
<td>Stores two DP FP values in a to p. The address p need not be 16-byte-aligned.</td>
</tr>
<tr>
<td>MOVUPS</td>
<td>__m128 _mm_loadu_ps(float * p)</td>
<td>Loads four SP FP values. The address need not be 16-byte-aligned.</td>
</tr>
<tr>
<td></td>
<td>void_mm_storeu_ps(float *p, __m128 a)</td>
<td>Stores four SP FP values. The address need not be 16-byte-aligned.</td>
</tr>
<tr>
<td>MULPD</td>
<td>__m128d _mm_mulu_pd(__m128d a, __m128d b)</td>
<td>Multiplies the two DP FP values of a and b.</td>
</tr>
<tr>
<td>MULPS</td>
<td>__m128d _mm_mul_ssd(__m128d a, __m128d b)</td>
<td>Multiplies the four SP FP values of a and b.</td>
</tr>
<tr>
<td>MULSD</td>
<td>__m128d _mm_mul_sd(__m128d a, __m128d b)</td>
<td>Multiplies the lower DP FP value of a and b; the upper DP FP value are passed through from a.</td>
</tr>
<tr>
<td>MULSS</td>
<td>__m128 _mm_mul_ssd(__m128 a, __m128 b)</td>
<td>Multiplies the lower SP FP value of a and b; the upper three SP FP values are passed through from a.</td>
</tr>
<tr>
<td>MWAIT</td>
<td>void_mm_mwait(unsigned extensions, unsigned hints)</td>
<td>A hint that allows the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events.</td>
</tr>
<tr>
<td>ORPD</td>
<td>__m128d _mm_or_pd(__m128d a, __m128d b)</td>
<td>Computes the bitwise OR of the two DP FP values of a and b.</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ORPS</td>
<td>_m128_mm_or_ps(___m128 a, ___m128 b)</td>
<td>Computes the bitwise OR of the four SP FP values of a and b.</td>
</tr>
<tr>
<td>PACKSSWB</td>
<td>_m128i_mm_packs_epi16(___m128i m1, ___m128i m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PACKSSWB</td>
<td>__m64_mm_packs_pi16(__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PACKSSDW</td>
<td>_m128i_mm_packs_epi32(___m128i m1, ___m128i m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PACKSSDW</td>
<td>__m64_mm_packs_pi32(__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PACKUSWB</td>
<td>_m128i_mm_packus_epi16(___m128i m1, ___m128i m2)</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PACKUSWB</td>
<td>__m64 __m_mm Packspu16(__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PADDB</td>
<td>__m128i __m_mm add_epi8(__m128i m1, __m128i m2)</td>
<td>Add the 16 8-bit values in m1 to the 16 8-bit values in m2.</td>
</tr>
<tr>
<td>PADDB</td>
<td>__m64 __m_mm add_pi8(__m64 m1, __m64 m2)</td>
<td>Add the eight 8-bit values in m1 to the eight 8-bit values in m2.</td>
</tr>
<tr>
<td>PADDW</td>
<td>__m128i __m_mm addw_epi16(__m128i m1, __m128i m2)</td>
<td>Add the 8 16-bit values in m1 to the 8 16-bit values in m2.</td>
</tr>
<tr>
<td>PADDW</td>
<td>__m64 __m_mm addw_pi16(__m64 m1, __m64 m2)</td>
<td>Add the four 16-bit values in m1 to the four 16-bit values in m2.</td>
</tr>
<tr>
<td>PADD</td>
<td>__m128i __m_mm add_epi32(__m128i m1, __m128i m2)</td>
<td>Add the 4 32-bit values in m1 to the 4 32-bit values in m2.</td>
</tr>
<tr>
<td>PADD</td>
<td>__m64 __m_mm add_pi32(__m64 m1, __m64 m2)</td>
<td>Add the two 32-bit values in m1 to the two 32-bit values in m2.</td>
</tr>
<tr>
<td>PADDQ</td>
<td>__m128i __m_mm add_epi64(__m128i m1, __m128i m2)</td>
<td>Add the 2 64-bit values in m1 to the 2 64-bit values in m2.</td>
</tr>
<tr>
<td>PADDQ</td>
<td>__m64 __m_mm add_si64(__m64 m1, __m64 m2)</td>
<td>Add the 64-bit value in m1 to the 64-bit value in m2.</td>
</tr>
<tr>
<td>PADDSB</td>
<td>__m128i __m_mm add_epi8(__m128i m1, __m128i m2)</td>
<td>Add the 16 signed 8-bit values in m1 to the 16 signed 8-bit values in m2 and saturate.</td>
</tr>
<tr>
<td>PADDSB</td>
<td>__m64 __m_mm add_pi8(__m64 m1, __m64 m2)</td>
<td>Add the eight signed 8-bit values in m1 to the eight signed 8-bit values in m2 and saturate.</td>
</tr>
<tr>
<td>PADDSW</td>
<td>__m128i __m_mm addw_epi16(__m128i m1, __m128i m2)</td>
<td>Add the 8 signed 16-bit values in m1 to the 8 signed 16-bit values in m2 and saturate.</td>
</tr>
<tr>
<td>PADDSW</td>
<td>__m64 __m_mm addw_pi16(__m64 m1, __m64 m2)</td>
<td>Add the four signed 16-bit values in m1 to the four signed 16-bit values in m2 and saturate.</td>
</tr>
</tbody>
</table>
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PADDUSB</td>
<td>_m128i_mm_adds_epu8(__m128i m1, __m128i m2)</td>
<td>Add the 16 unsigned 8-bit values in m1 to the 16 unsigned 8-bit values in m2 and saturate.</td>
</tr>
<tr>
<td>PADDUSB</td>
<td>_m64_mm_adds_pu8(__m64 m1, __m64 m2)</td>
<td>Add the eight unsigned 8-bit values in m1 to the eight unsigned 8-bit values in m2 and saturate.</td>
</tr>
<tr>
<td>PADDUSW</td>
<td>_m128i_mm_adds_epu16(__m128i m1, __m128i m2)</td>
<td>Add the 8 unsigned 16-bit values in m1 to the eight unsigned 8-bit values in m2 and saturate.</td>
</tr>
<tr>
<td>PADDUSW</td>
<td>_m64_mm_adds_pu16(__m64 m1, __m64 m2)</td>
<td>Add the four unsigned 16-bit values in m1 to the four unsigned 16-bit values in m2 and saturate.</td>
</tr>
<tr>
<td>PAND</td>
<td>_m128i_mm_and_si128(__m128i m1, __m128i m2)</td>
<td>Perform a bitwise AND of the 128-bit value in m1 with the 128-bit value in m2.</td>
</tr>
<tr>
<td>PAND</td>
<td>_m64_mm_and_si64(__m64 m1, __m64 m2)</td>
<td>Perform a bitwise AND of the 64-bit value in m1 with the 64-bit value in m2.</td>
</tr>
<tr>
<td>PANDN</td>
<td>_m128i_mm_andnot_si128(__m128i m1, __m128i m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PANDN</td>
<td>_m64_mm_andnot_si64(__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PAUSE</td>
<td>void_mm_pause(void)</td>
<td>The execution of the next instruction is delayed by an implementation-specific amount of time. No architectural state is modified.</td>
</tr>
<tr>
<td>PAVGB</td>
<td>_m128i_mm_avg_epu8(__m128i a, __m128i b)</td>
<td>Perform the packed average on the 16 8-bit values of the two operands.</td>
</tr>
<tr>
<td>PAVGB</td>
<td>_m64_mm_avg_pu8(__m64 a, __m64 b)</td>
<td>Perform the packed average on the 8 8-bit values of the two operands.</td>
</tr>
<tr>
<td>PAVGW</td>
<td>_m128i_mm_avg_epu16(__m128i a, __m128i b)</td>
<td>Perform the packed average on the 8 16-bit values of the two operands.</td>
</tr>
</tbody>
</table>
## Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAVGW</td>
<td>__m64_mm_avg_pu16(__m64 a, __m64 b)</td>
<td>Perform the packed average on the four 16-bit values of the two operands.</td>
</tr>
<tr>
<td>PCMPEQB</td>
<td>__m128i_mm_cmpeq_epi8(__m128i m1, __m128i m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PCMPEQB</td>
<td>__m64_mm_cmpeq_pi8(__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PCMPEQW</td>
<td>__m128i_mm_cmpeq_epi16 (__m128i m1, __m128i m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PCMPEQW</td>
<td>__m64_mm_cmpeq_pi16 (__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PCMPEQD</td>
<td>__m128i_mm_cmpeq_epi32(__m128i m1, __m128i m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PCMPEQD</td>
<td>__m64_mm_cmpeq_pi32(__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCMPGTB</td>
<td>_m128i _mm_cmpgt_epi8 (__m128i m1, __m128i m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PCMPGTB</td>
<td>_m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PCMPGTW</td>
<td>_m128i _mm_cmpgt_epi16(__m128i m1, __m128i m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PCMPGTW</td>
<td>_m64 _mm_cmpgt_pi16 (__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PCMPGTD</td>
<td>_m128i _mm_cmpgt_epi32(__m128i m1, __m128i m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PCMPGTD</td>
<td>_m64 _mm_cmpgt_pi32(__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PEXTRW</td>
<td>int _mm_extract_epi16(__m128i a, int n)</td>
<td>Extracts one of the 8 words of a. The selector n must be an immediate.</td>
</tr>
<tr>
<td>PEXTRW</td>
<td>int _mm_extract_pi16(__m64 a, int n)</td>
<td>Extracts one of the four words of a. The selector n must be an immediate.</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PINSRW</td>
<td>__m128i __mm_insert_epi16(__m128i a, int d, int n)</td>
<td>Inserts word d into one of 8 words of a. The selector n must be an immediate.</td>
</tr>
<tr>
<td>PINSRW</td>
<td>__m64 __mm_insert_pi16(__m64 a, int d, int n)</td>
<td>Inserts word d into one of four words of a. The selector n must be an immediate.</td>
</tr>
<tr>
<td>PMADDWD</td>
<td>__m128i __mm_madd_epi16(__m128i m1 __m128i m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PMADDWD</td>
<td>__m64 __mm_madd_pi16(__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PMAXSW</td>
<td>__m128i __mm_max_epi16(__m128i a, __m128i b)</td>
<td>Computes the element-wise maximum of the 16-bit integers in a and b.</td>
</tr>
<tr>
<td>PMAXSW</td>
<td>__m64 __mm_max_pi16(__m64 a, __m64 b)</td>
<td>Computes the element-wise maximum of the 32-bit integers in a and b.</td>
</tr>
<tr>
<td>PMAXUB</td>
<td>__m128i __mm_max_epu8(__m128i a, __m128i b)</td>
<td>Computes the element-wise maximum of the unsigned bytes in a and b.</td>
</tr>
<tr>
<td>PMAXUB</td>
<td>__m64 __mm_max_pu8(__m64 a, __m64 b)</td>
<td>Computes the element-wise maximum of the unsigned bytes in a and b.</td>
</tr>
<tr>
<td>PMINSW</td>
<td>__m128i __mm_min_epi16(__m128i a, __m128i b)</td>
<td>Computes the element-wise minimum of the 16-bit integers in a and b.</td>
</tr>
<tr>
<td>PMINSW</td>
<td>__m64 __mm_min_pi16(__m64 a, __m64 b)</td>
<td>Computes the element-wise minimum of the 32-bit integers in a and b.</td>
</tr>
<tr>
<td>PMINUB</td>
<td>__m128i __mm_min_epu8(__m128i a, __m128i b)</td>
<td>Computes the element-wise minimum of the unsigned bytes in a and b.</td>
</tr>
<tr>
<td>PMINUB</td>
<td>__m64 __mm_min_pu8(__m64 a, __m64 b)</td>
<td>Computes the element-wise minimum of the unsigned bytes in a and b.</td>
</tr>
<tr>
<td>PMOVMSKB</td>
<td>int __mm_movemask_epi8(__m128i a)</td>
<td>Creates an 16-bit mask from the most significant bits of the bytes in a.</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PMOVMSKB</td>
<td>int _mm_movemask_pi8(__m64 a)</td>
<td>Creates an 8-bit mask from the most significant bits of the bytes in a.</td>
</tr>
<tr>
<td>PMULHUW</td>
<td>_m128i _mm_mulhi_epi16(__m128i a, __m128i b)</td>
<td>Multiplies the 8 unsigned words in a and b, returning the upper 16 bits of the eight 32-bit intermediate results in packed form.</td>
</tr>
<tr>
<td>PMULHUW</td>
<td>_m64 _mm_mulhi_pu16(__m64 a, __m64 b)</td>
<td>Multiplies the 4 unsigned words in a and b, returning the upper 16 bits of the four 32-bit intermediate results in packed form.</td>
</tr>
<tr>
<td>PMULHW</td>
<td>_m128i _mm_mulhi_epi16(__m128i m1, __m128i m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PMULHW</td>
<td>_m64 _mm_mulhi_pil16(__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PMULLW</td>
<td>_m128i _mm_mullo_epi16(__m128i m1, __m128i m2)</td>
<td>Multiply 8 16-bit values in m1 by 8 16-bit values in m2 and produce the low 16 bits of the 8 results.</td>
</tr>
<tr>
<td>PMULLW</td>
<td>_m64 _mm_mullo_pil16(__m64 m1, __m64 m2)</td>
<td>Multiply four 16-bit values in m1 by four 16-bit values in m2 and produce the low 16 bits of the four results.</td>
</tr>
<tr>
<td>PMULUDQ</td>
<td>_m64 _mm_mul_su32(__m64 m1, __m64 m2)</td>
<td>Multiply lower 32-bit unsigned value in m1 by the lower 32-bit unsigned value in m2 and store the 64 bit results.</td>
</tr>
<tr>
<td></td>
<td>_m128i _mm_mul_epu32(__m128i m1, __m128i m2)</td>
<td>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.</td>
</tr>
<tr>
<td>POR</td>
<td>_m64 _mm_or_si64(__m64 m1, __m64 m2)</td>
<td>Perform a bitwise OR of the 64-bit value in m1 with the 64-bit value in m2.</td>
</tr>
<tr>
<td>POR</td>
<td>_m128i _mm_or_si128(__m128i m1, __m128i m2)</td>
<td>Perform a bitwise OR of the 128-bit value in m1 with the 128-bit value in m2.</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PREFETCHh</td>
<td>void _mm_prefetch(char *a, int sel)</td>
<td>Loads one cache line of data from address p to a location &quot;closer&quot; to the processor. The value sel specifies the type of prefetch operation.</td>
</tr>
<tr>
<td>PSADBW</td>
<td>__m128i _mm_sad_epu8(__m128i a, __m128i b)</td>
<td>Compute the absolute differences of the 16 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.</td>
</tr>
<tr>
<td>PSADBW</td>
<td>__m64 _mm_sad_pu8(__m64 a, __m64 b)</td>
<td>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.</td>
</tr>
<tr>
<td>PSUFD</td>
<td>__m128i _mm_shuffle_epi32(__m128i a, int n)</td>
<td>Returns a combination of the four doublewords of a. The selector n must be an immediate.</td>
</tr>
<tr>
<td>PSUFW6</td>
<td>__m128i _mm_shufflehi_epi16(__m128i a, int n)</td>
<td>Shuffle the upper four 16-bit words in a as specified by n. The selector n must be an immediate.</td>
</tr>
<tr>
<td>PSUFWLW</td>
<td>__m128i _mm_shufflelo_epi16(__m128i a, int n)</td>
<td>Shuffle the lower four 16-bit words in a as specified by n. The selector n must be an immediate.</td>
</tr>
<tr>
<td>PSUFW</td>
<td>__m64 _mm_shuffle_pi16(__m64 a, int n)</td>
<td>Returns a combination of the four words of a. The selector n must be an immediate.</td>
</tr>
<tr>
<td>PSLLW</td>
<td>__m128i _mm_sll_epi16(__m128i m, __m128i count)</td>
<td>Shift each of 8 16-bit values in m left the amount specified by count while shifting in zeroes.</td>
</tr>
<tr>
<td>PSLLW</td>
<td>__m128i _mm_slli_epi16(__m128i m, int count)</td>
<td>Shift each of 8 16-bit values in m left the amount specified by count while shifting in zeroes.</td>
</tr>
<tr>
<td>PSLLW</td>
<td>__m64 _mm_sll_pi16(__m64 m, __m64 count)</td>
<td>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.</td>
</tr>
</tbody>
</table>
Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>__m64 _mm_slli_pi16(__m64 m, int count)</td>
<td>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.</td>
</tr>
<tr>
<td>PSLLD</td>
<td>__m128i _mm_slli_epi32(__m128i m, int count)</td>
<td>Shift each of 4 32-bit values in m left the amount specified by count while shifting in zeroes.</td>
</tr>
<tr>
<td></td>
<td>__m128i _mm_slli_epi32(__m128i m, __m128i count)</td>
<td>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.</td>
</tr>
<tr>
<td>PSLLD</td>
<td>__m64 _mm_slli_pi32(__m64 m, int count)</td>
<td>Shift two 32-bit values in m left the amount specified by count while shifting in zeroes.</td>
</tr>
<tr>
<td></td>
<td>__m64 _mm_slli_pi32(__m64 m, __m64 count)</td>
<td>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.</td>
</tr>
<tr>
<td>PSLLQ</td>
<td>__m64 _mm_sll_epi64(__m64 m, __m64 count)</td>
<td>Shift the 64-bit value in m left the amount specified by count while shifting in zeroes.</td>
</tr>
<tr>
<td></td>
<td>__m64 _mm_sll_epi64(__m64 m, int count)</td>
<td>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.</td>
</tr>
<tr>
<td>PSLLQ</td>
<td>__m128i _mm_sll_epi64(__m128i m, __m128i count)</td>
<td>Shift each of two 64-bit values in m left by the amount specified by count while shifting in zeroes.</td>
</tr>
<tr>
<td></td>
<td>__m128i _mm_sll_epi64(__m128i m, int count)</td>
<td>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.</td>
</tr>
<tr>
<td>PSLLDQ</td>
<td>__m128i _mm_slli_si128(__m128i m, int imm)</td>
<td>Shift 128 bit in m left by imm bytes while shifting in zeroes.</td>
</tr>
<tr>
<td>PSRAW</td>
<td>__m128i _mm_sra_epi16(__m128i m, __m128i count)</td>
<td>Shift each of 8 16-bit values in m right the amount specified by count while shifting in the sign bit.</td>
</tr>
</tbody>
</table>
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSRAW</td>
<td>__m64 _mm_sra_pi16(__m64 m, __m64 count)</td>
<td>Shift four 16-bit values in m right the amount specified by count while shifting in the sign bit.</td>
</tr>
<tr>
<td></td>
<td>__m64 _mm_srai_pi16(__m64 m, int count)</td>
<td>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.</td>
</tr>
<tr>
<td>PSRAD</td>
<td>__m128i _mm_sra_epi32 (__m128i m, __m128i count)</td>
<td>Shift each of 4 32-bit values in m right the amount specified by count while shifting in the sign bit.</td>
</tr>
<tr>
<td></td>
<td>__m128i _mm_srai_epi32 (__m128i m, int count)</td>
<td>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.</td>
</tr>
<tr>
<td>PSRAD</td>
<td>__m64 _mm_sra_pi32 (__m64 m, __m64 count)</td>
<td>Shift two 32-bit values in m right the amount specified by count while shifting in the sign bit.</td>
</tr>
<tr>
<td></td>
<td>__m64 _mm_srai_pi32 (__m64 m, int count)</td>
<td>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.</td>
</tr>
<tr>
<td>PSRLW</td>
<td>__m128i _mm_srl_epi16 (__m128i m, __m128i count)</td>
<td>Shift each of 8 16-bit values in m right the amount specified by count while shifting in zeroes.</td>
</tr>
<tr>
<td></td>
<td>__m128i _mm_slli_epi16 (__m128i m, int count)</td>
<td>Shift each of 8 16-bit values in m right the amount specified by count while shifting in zeroes.</td>
</tr>
<tr>
<td>PSRLW</td>
<td>__m64 _mm_srl_pi16 (__m64 m, __m64 count)</td>
<td>Shift four 16-bit values in m right the amount specified by count while shifting in zeroes.</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>PSRLD</td>
<td>__m128i _mm_srl_epi32 (__m128i m, __m128i count)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>__m128i _mm_srli_epi32 (__m128i m, int count)</td>
<td>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.</td>
</tr>
<tr>
<td>PSRLD</td>
<td>__m64 _mm_srl_pi32 (__m64 m, __m64 count)</td>
<td>Shift two 32-bit values in m right the amount specified by count while shifting in zeroes.</td>
</tr>
<tr>
<td></td>
<td>__m64 _mm_srli_pi32 (__m64 m, int count)</td>
<td>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.</td>
</tr>
<tr>
<td>PSRLQ</td>
<td>__m128i _mm_srl_epi64 (__m128i m, __m128i count)</td>
<td>Shift the 2 64-bit value in m right the amount specified by count while shifting in zeroes.</td>
</tr>
<tr>
<td></td>
<td>__m128i _mm_srli_epi64 (__m128i m, int count)</td>
<td>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.</td>
</tr>
<tr>
<td>PSRLQ</td>
<td>__m64 _mm_srl_si64 (__m64 m, __m64 count)</td>
<td>Shift the 64-bit value in m right the amount specified by count while shifting in zeroes.</td>
</tr>
<tr>
<td></td>
<td>__m64 _mm_srli_si64 (__m64 m, int count)</td>
<td>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.</td>
</tr>
<tr>
<td>PSRLDQ</td>
<td>__m128i _mm_srli_si128(__m128i m, int imm)</td>
<td>Shift 128 bit in m right by imm bytes while shifting in zeroes.</td>
</tr>
<tr>
<td>PSUBB</td>
<td>__m128i _mm_sub_epi8(__m128i m1, __m128i m2)</td>
<td>Subtract the 16 8-bit values in m2 from the 16 8-bit values in m1.</td>
</tr>
<tr>
<td>PSUBB</td>
<td>__m64 _mm_sub_pi8(__m64 m1, __m64 m2)</td>
<td>Subtract the eight 8-bit values in m2 from the eight 8-bit values in m1.</td>
</tr>
</tbody>
</table>
Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSUBW</td>
<td>__m128i__mm_sub_epi16(__m128i m1, __m128i m2)</td>
<td>Subtract the 8 16-bit values in m2 from the 8 16-bit values in m1.</td>
</tr>
<tr>
<td>PSUBW</td>
<td>__m64__mm_sub_pi16(__m64 m1, __m64 m2)</td>
<td>Subtract the four 16-bit values in m2 from the four 16-bit values in m1.</td>
</tr>
<tr>
<td>PSUBD</td>
<td>__m128i__mm_sub_epi32(__m128i m1, __m128i m2)</td>
<td>Subtract the 4 32-bit values in m2 from the 4 32-bit values in m1.</td>
</tr>
<tr>
<td>PSUBD</td>
<td>__m64__mm_sub_pi32(__m64 m1, __m64 m2)</td>
<td>Subtract the two 32-bit values in m2 from the two 32-bit values in m1.</td>
</tr>
<tr>
<td>PSUBQ</td>
<td>__m128i__mm_sub_epi64(__m128i m1, __m128i m2)</td>
<td>Subtract the 2 64-bit values in m2 from the 2 64-bit values in m1.</td>
</tr>
<tr>
<td>PSUBQ</td>
<td>__m64__mm_sub_si64(__m64 m1, __m64 m2)</td>
<td>Subtract the 64-bit values in m2 from the 64-bit values in m1.</td>
</tr>
<tr>
<td>PSUBSB</td>
<td>__m128i__mm_subs_epi8(__m128i m1, __m128i m2)</td>
<td>Subtract the 16 signed 8-bit values in m2 from the 16 signed 8-bit values in m1 and saturate.</td>
</tr>
<tr>
<td>PSUBSB</td>
<td>__m64__mm_subs_pi8(__m64 m1, __m64 m2)</td>
<td>Subtract the eight signed 8-bit values in m2 from the eight signed 8-bit values in m1 and saturate.</td>
</tr>
<tr>
<td>PSUBSW</td>
<td>__m128i__mm_subs_epi16(__m128i m1, __m128i m2)</td>
<td>Subtract the 8 signed 16-bit values in m2 from the 8 signed 16-bit values in m1 and saturate.</td>
</tr>
<tr>
<td>PSUBSW</td>
<td>__m64__mm_subs_pi16(__m64 m1, __m64 m2)</td>
<td>Subtract the four signed 16-bit values in m2 from the four signed 16-bit values in m1 and saturate.</td>
</tr>
<tr>
<td>PSUBUSB</td>
<td>__m128i__mm_sub_epu8(__m128i m1, __m128i m2)</td>
<td>Subtract the 16 unsigned 8-bit values in m2 from the 16 unsigned 8-bit values in m1 and saturate.</td>
</tr>
<tr>
<td>PSUBUSB</td>
<td>__m64__mm_sub_pu8(__m64 m1, __m64 m2)</td>
<td>Subtract the eight unsigned 8-bit values in m2 from the eight unsigned 8-bit values in m1 and saturate.</td>
</tr>
<tr>
<td>PSUBUSW</td>
<td>__m128i__mm_sub_epu16(__m128i m1, __m128i m2)</td>
<td>Subtract the 8 unsigned 16-bit values in m2 from the 8 unsigned 16-bit values in m1 and saturate.</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PSUBUSW</td>
<td>__m64_mm_sub_pu16(__m64 m1, __m64 m2)</td>
<td>Subtract the four unsigned 16-bit values in m2 from the four unsigned 16-bit values in m1 and saturate.</td>
</tr>
<tr>
<td>PUNPCKHBW</td>
<td>__m64_mm_unpackhi_pi8(__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PUNPCKHBW</td>
<td>__m128i_mm_unpackhi_epi8(__m128i m1, __m128i m2)</td>
<td>Interleave the 8 8-bit values from the high half of m1 with the 8 values from the high half of m2.</td>
</tr>
<tr>
<td>PUNPCKHWD</td>
<td>__m64_mm_unpackhi_pl16(__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PUNPCKHWD</td>
<td>__m128i_mm_unpackhi_epi16(__m128i m1, __m128i m2)</td>
<td>Interleave the 4 16-bit values from the high half of m1 with the 4 values from the high half of m2.</td>
</tr>
<tr>
<td>PUNPCKHDQ</td>
<td>__m64_mm_unpackhi_pl32(__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PUNPCKHDQ</td>
<td>__m128i_mm_unpackhi_epi32(__m128i m1, __m128i m2)</td>
<td>Interleave two 32-bit value from the high half of m1 with the two 32-bit value from the high half of m2.</td>
</tr>
<tr>
<td>PUNPCKHDQ</td>
<td>__m128i_mm_unpackhi_epi64(__m128i m1, __m128i m2)</td>
<td>Interleave the 64-bit value from the high half of m1 with the 64-bit value from the high half of m2.</td>
</tr>
<tr>
<td>PUNPCKLBW</td>
<td>__m64_mm_unpacklo_pi8 (__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PUNPCKLBW</td>
<td>__m128i_mm_unpacklo_epi8 (__m128i m1, __m128i m2)</td>
<td>Interleave the 8 8-bit values from the low half of m1 with the 8 values from the low half of m2.</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PUNPCKLWD</td>
<td>__m64 _mm_unpacklo_pi16(__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PUNPCKLWD</td>
<td>__m128i _mm_unpacklo_epi16(__m128i m1, __m128i m2)</td>
<td>Interleave the 4 16-bit values from the low half of m1 with the 4 values from the low half of m2.</td>
</tr>
<tr>
<td>PUNPCKLDQ</td>
<td>__m64 _mm_unpacklo_pi32(__m64 m1, __m64 m2)</td>
<td>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.</td>
</tr>
<tr>
<td>PUNPCKLDQ</td>
<td>__m128i _mm_unpacklo_epi32(__m128i m1, __m128i m2)</td>
<td>Interleave two 32-bit value from the low half of m1 with the two 32-bit value from the low half of m2.</td>
</tr>
<tr>
<td>PUNPCKLDQ</td>
<td>__m128i _mm_unpacklo_epi64(__m128i m1, __m128i m2)</td>
<td>Interleave the 64-bit value from the low half of m1 with the 64-bit value from the low half of m2.</td>
</tr>
<tr>
<td>PXOR</td>
<td>__m64 _mm_xor_si64(__m64 m1, __m64 m2)</td>
<td>Perform a bitwise XOR of the 64-bit value in m1 with the 64-bit value in m2.</td>
</tr>
<tr>
<td>PXOR</td>
<td>__m128i _mm_xor_si128(__m128i m1, __m128i m2)</td>
<td>Perform a bitwise XOR of the 128-bit value in m1 with the 128-bit value in m2.</td>
</tr>
<tr>
<td>RCPPS</td>
<td>__m128 _mm_rcp_ps(__m128 a)</td>
<td>Computes the approximations of the reciprocals of the four SP FP values of a.</td>
</tr>
<tr>
<td>RCPSS</td>
<td>__m128 _mm_rcp_ss(__m128 a)</td>
<td>Computes the approximation of the reciprocal of the lower SP FP value of a; the upper three SP FP values are passed through.</td>
</tr>
<tr>
<td>RSQRTPS</td>
<td>__m128 _mm_rsqrtps(__m128 a)</td>
<td>Computes the approximations of the reciprocals of the square roots of the four SP FP values of a.</td>
</tr>
<tr>
<td>RSQRTSS</td>
<td>__m128 _mm_rsqrts(__m128 a)</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFENCE</td>
<td>void_mm_sfence(void)</td>
<td>Guarantees that every preceding store is globally visible before any subsequent store.</td>
</tr>
<tr>
<td>SHUFPD</td>
<td>__m128d __mm_shuffle_pd(__m128d a, __m128d b, unsigned int imm8)</td>
<td>Selects two specific DP FP values from a and b, based on the mask imm8. The mask must be an immediate.</td>
</tr>
<tr>
<td>SHUFPS</td>
<td>__m128 __mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)</td>
<td>Selects four specific SP FP values from a and b, based on the mask imm8. The mask must be an immediate.</td>
</tr>
<tr>
<td>SQRTPD</td>
<td>__m128d __mm_sqrt_pd(__m128d a)</td>
<td>Computes the square roots of the two DP FP values of a.</td>
</tr>
<tr>
<td>SQRTPS</td>
<td>__m128 __mm_sqrt_ps(__m128 a)</td>
<td>Computes the square roots of the four SP FP values of a.</td>
</tr>
<tr>
<td>SQRTSD</td>
<td>__m128d __mm_sqrt_sd(__m128d a)</td>
<td>Computes the square root of the lower DP FP value of a; the upper DP FP values are passed through.</td>
</tr>
<tr>
<td>SQRTSS</td>
<td>__m128 __mm_sqrt_ss(__m128 a)</td>
<td>Computes the square root of the lower SP FP value of a; the upper three SP FP values are passed through.</td>
</tr>
<tr>
<td>STMXCSR</td>
<td>__mm_getcsr(void)</td>
<td>Returns the contents of the control register.</td>
</tr>
<tr>
<td>SUBPD</td>
<td>__m128d __mm_sub_pd(__m128d a, __m128d b)</td>
<td>Subtracts the two DP FP values of a and b.</td>
</tr>
<tr>
<td>SUBPS</td>
<td>__m128 __mm_sub_ps(__m128 a, __m128 b)</td>
<td>Subtracts the four SP FP values of a and b.</td>
</tr>
<tr>
<td>SUBSD</td>
<td>__m128d __mm_sub_sd(__m128d a, __m128d b)</td>
<td>Subtracts the lower DP FP values of a and b. The upper DP FP values are passed through from a.</td>
</tr>
<tr>
<td>SUBSS</td>
<td>__m128 __mm_sub_ss(__m128 a, __m128 b)</td>
<td>Subtracts the lower SP FP values of a and b. The upper three SP FP values are passed through from a.</td>
</tr>
<tr>
<td>UCOMISD</td>
<td>int __mm_ucomieq_sd(__m128d a, __m128d b)</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCOMISS</td>
<td>int _mm_ucomieq_ss(__m128 a, __m128 b)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>int _mm_ucomilt_ss(__m128 a, __m128 b)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>int _mm_ucomile_ss(__m128 a, __m128 b)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>int _mm_ucomigt_ss(__m128 a, __m128 b)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>int _mm_ucomige_ss(__m128 a, __m128 b)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>int _mm_ucomineq_ss(__m128 a, __m128 b)</td>
<td>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.</td>
</tr>
</tbody>
</table>

**INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL**

- **_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.
- **_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.
- **_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.
- **_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.
- **_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.
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>int _mm_ucomige_ss(__m128 a, __m128 b)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>int _mm_ucomineq_ss(__m128 a, __m128 b)</td>
<td>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.</td>
</tr>
<tr>
<td>UNPCKHPD</td>
<td>__m128d _mm_unpackhi_pd(__m128d a, __m128d b)</td>
<td>Selects and interleaves the upper DP FP values from a and b.</td>
</tr>
<tr>
<td>UNPCKHPS</td>
<td>__m128 _mm_unpackhi_ps(__m128 a, __m128 b)</td>
<td>Selects and interleaves the upper two SP FP values from a and b.</td>
</tr>
<tr>
<td>UNPCKLPD</td>
<td>__m128d _mm_unpacklo_pd(__m128d a, __m128d b)</td>
<td>Selects and interleaves the lower DP FP values from a and b.</td>
</tr>
<tr>
<td>UNPCKLPS</td>
<td>__m128 _mm_unpacklo_ps(__m128 a, __m128 b)</td>
<td>Selects and interleaves the lower two SP FP values from a and b.</td>
</tr>
<tr>
<td>XORPD</td>
<td>__m128d _mm_xor_pd(__m128d a, __m128d b)</td>
<td>Computes bitwise EXOR (exclusive-or) of the two DP FP values of a and b.</td>
</tr>
<tr>
<td>XORPS</td>
<td>__m128 _mm_xor_ps(__m128 a, __m128 b)</td>
<td>Computes bitwise EXOR (exclusive-or) of the four SP FP values of a and b.</td>
</tr>
</tbody>
</table>
## C.2. COMPOSITE INTRINSICS

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(composite)</td>
<td>__m128i __mm_set_epi64(__m64 q1, __m64 q0)</td>
<td>Sets the two 64-bit values to the two inputs.</td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128i __mm_set_epi32(int i3, int i2, int i1, int i0)</td>
<td>Sets the 4 32-bit values to the 4 inputs.</td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128i __mm_set_epi16(short w7, short w6, short w5, short w4, short w3,</td>
<td>Sets the 8 16-bit values to the 8 inputs.</td>
</tr>
<tr>
<td></td>
<td>short w2, short w1)</td>
<td></td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128i __mm_set_epi8(char w15, char w14, char w13, char w12, char w11,</td>
<td>Sets the 16 8-bit values to the 16 inputs.</td>
</tr>
<tr>
<td></td>
<td>char w10, char w9, char w8, char w7, char w6, char w5, char w4, char w3,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>char w2, char w1, char w0)</td>
<td></td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128i __mm_setl_epi64(__m64 q)</td>
<td>Sets the 2 64-bit values to the input.</td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128i __mm_setl_epi32(int a)</td>
<td>Sets the 4 32-bit values to the input.</td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128i __mm_setl_epi16(short a)</td>
<td>Sets the 8 16-bit values to the input.</td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128i __mm_setl_epi8(char a)</td>
<td>Sets the 16 8-bit values to the input.</td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128i __mm_setr_epi64(__m64 q1, __m64 q0)</td>
<td>Sets the two 64-bit values to the two inputs in reverse order.</td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128i __mm_setr_epi32(int i3, int i2, int i1, int i0)</td>
<td>Sets the 4 32-bit values to the 4 inputs in reverse order.</td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128i __mm_setr_epi16(short w7, short w6, short w5, short w4, short w3,</td>
<td>Sets the 8 16-bit values to the 8 inputs in reverse order.</td>
</tr>
<tr>
<td></td>
<td>short w2, short w1)</td>
<td></td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128i __mm_setr_epi8(char w15, char w14, char w13, char w12, char w11,</td>
<td>Sets the 16 8-bit values to the 16 inputs in reverse order.</td>
</tr>
<tr>
<td></td>
<td>char w10, char w9, char w8, char w7, char w6, char w5, char w4, char w3,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>char w2, char w1, char w0)</td>
<td></td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128i __mm_setzero_si128()</td>
<td>Sets all bits to 0.</td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128 __mm_set_ps1(float w)</td>
<td>Sets the four SP FP values to w.</td>
</tr>
<tr>
<td></td>
<td>__m128 __mm_set1_ps(float w)</td>
<td></td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128d __mm_set_sd(double w)</td>
<td>Sets the two DP FP values to w.</td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128d __mm_set_pd(double z, double y)</td>
<td>Sets the lower DP FP values to w.</td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128d __mm_set_pd(double z, double y)</td>
<td>Sets the two DP FP values to the two inputs.</td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128d __mm_set_ps(float z, float y, float x, float w)</td>
<td>Sets the four SP FP values to the four inputs.</td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128d __mm_setr_pd(double z, double y)</td>
<td>Sets the two DP FP values to the two inputs in reverse order.</td>
</tr>
</tbody>
</table>
### INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL

#### Table C-2. Composite Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(composite)</td>
<td>__m128_d_mm_setr_ps(float z, float y, float x, float w)</td>
<td>Sets the four SP FP values to the four inputs in reverse order.</td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128_d_mm_setzero_ps(void)</td>
<td>Clears the two DP FP values.</td>
</tr>
<tr>
<td>(composite)</td>
<td>__m128_d_mm_setzero_ps(void)</td>
<td>Clears the four SP FP values.</td>
</tr>
<tr>
<td>MOVSD + shuffle</td>
<td>__m128d_mm_load_pd(double * p)</td>
<td>Loads a single DP FP value, copying it into both DP FP values.</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_load1_pd(double *p)</td>
<td></td>
</tr>
<tr>
<td>MOVSS + shuffle</td>
<td>__m128_d_mm_load_ps1(float * p)</td>
<td>Loads a single SP FP value, copying it into all four words.</td>
</tr>
<tr>
<td></td>
<td>__m128_d_mm_load1_ps(float *p)</td>
<td></td>
</tr>
<tr>
<td>MOVAPD + shuffle</td>
<td>__m128_d_mm_loadr_pd(double * p)</td>
<td>Loads two DP FP values in reverse order. The address must be 16-byte-aligned.</td>
</tr>
<tr>
<td>MOVAPS + shuffle</td>
<td>__m128_d_mm_loadr_ps(float * p)</td>
<td>Loads four SP FP values in reverse order. The address must be 16-byte-aligned.</td>
</tr>
<tr>
<td>MOVSD + shuffle</td>
<td>void_mm_store1_pd(double *p, __m128d a)</td>
<td>Stores the lower DP FP value across both DP FP values.</td>
</tr>
<tr>
<td>MOVSS + shuffle</td>
<td>void_mm_store_ps1(float * p, __m128 a)</td>
<td>Stores the lower SP FP value across four words.</td>
</tr>
<tr>
<td>MOVAPD + shuffle</td>
<td>__mm_storer_pd(double * p, __m128d a)</td>
<td>Stores two DP FP values in reverse order. The address must be 16-byte-aligned.</td>
</tr>
<tr>
<td>MOVAPS + shuffle</td>
<td>__mm_storer_ps(float * p, __m128 a)</td>
<td>Stores four SP FP values in reverse order. The address must be 16-byte-aligned.</td>
</tr>
</tbody>
</table>
Index
INDEX FOR VOLUME 2A & 2B

A
AAA instruction ...................................... 3-16
AAD instruction ....................................... 3-17
AAM instruction ....................................... 3-18
AAS instruction ....................................... 3-19
Abbreviations, opcode key .......................... A-1
Access rights, segment descriptor ................. 3-392
ADC instruction ....................................... 3-20, 3-416
ADD instruction ....................................... 3-16, 3-20, 3-22, 3-188, 3-416
ADDPD instruction .................................... 3-24
ADDPDS instruction ................................... 3-26
Addressing methods
  codes .................................................. A-2
  operand codes ...................................... A-3
  register codes ..................................... A-3
Addressing, segments ................................ 1-5
ADDS instruction ...................................... 3-28
ADDSUBPD instruction ................................ 3-32
ADDSUBPS instruction ................................ 3-35
AND instruction ....................................... 3-38, 3-416
ANDNPD instruction .................................. 3-44
ANDNPS instruction .................................. 3-46
ANPD instruction ...................................... 3-40
ANDPS instruction .................................... 3-42
Arctangent, x87 FPU operation ....................... 3-265
ARPL instruction ..................................... 3-48

B
B (default stack size) flag, segment descriptor 4-83, 4-141
Base (operand addressing) ......................... 2-4
BCD integers
  packed .............................................. 3-188, 3-190, 3-215, 3-217
  unpacked ........................................... 3-16, 3-17, 3-18, 3-19
Binary numbers ..................................... 1-4
Bit order .............................................. 1-2
BOUND instruction .................................. 3-50
BOUND range exceeded exception (BR) ............ 3-50
Branch hints ......................................... 2-2
BSF instruction ....................................... 3-52
BSR instruction ....................................... 3-54
BSWAP instruction ................................... 3-56
BT instruction ........................................ 3-57
BTC instruction ...................................... 3-59, 3-416
BTR instruction ...................................... 3-61, 3-416
BTS instruction ...................................... 3-63, 3-416
Byte order ............................................ 1-2

C
Caches, invalidating (flushing) ..................... 3-370, 4-266
Call gate ............................................. 3-388
CALL instruction .................................... 3-65
CBW instruction ...................................... 3-76
CDQ instruction ...................................... 3-186
CF (carry) flag, EFLAGS register .................. 3-20, 3-22, 3-57, 3-59, 3-61, 3-63, 3-78, 3-85, 3-192, 3-350, 3-354, 3-525, 4-151, 4-184, 4-195, 4-197, 4-218, 4-229
Classify floating-point value, x87 FPU operation ........................................... 3-311
CLC instruction ....................................... 3-78
CLD instruction ....................................... 3-79
CLFLUSH instruction ................................ 3-80
CLI instruction ...................................... 3-82
CLTS instruction ..................................... 3-84
CMC instruction ...................................... 3-85
CMOVcc instructions ................................ 3-86
CMP instruction ...................................... 3-89
CMPPPD instruction ................................ 3-91
CMPFPS instruction ................................ 3-95
CMPS instruction .................................... 3-99, 4-164
CMPSB instruction .................................. 3-99
CMPSD instruction .................................. 3-99, 3-102
CMPSW instruction ................................ 3-106
CMPSXCHG instruction ............................... 3-110, 3-416
CMPSXCHG8B instruction ............................ 3-112
COMISS instruction ................................ 3-114
COMISS instruction ................................ 3-117
Compatibility
  software ............................................ 1-3
Condition code flags, EFLAGS register .......... 3-386
Condition code flags, x87 FPU status word
  flags affected by instructions .................. 3-12
  setting ............................................. 3-305, 3-307, 3-311
Conditional jump .................................... 3-380
Conforming code segment .......................... 3-387, 3-392
Constants (floating point), loading ............... 3-255
  Control registers, moving values to and from 3-462
  Cosine, x87 FPU operation ....................... 3-231, 3-285
  CPL ................................................. 3-82, 4-263
  CPUID instruction ................................ 3-120
  brand index ....................................... 3-126
  cache and TLB characteristics ................ 3-121, 3-132
  CLFLUSH instruction cache line size .......... 3-126
  extended function CPUID information .......... 3-122
  feature information ................................ 3-128
  local APIC physical ID ........................... 3-126
  processor brand string .......................... 3-122
  processor type fields ......................... 3-125
  version information .............................. 3-121
CR0 control register ................................ 4-208
CS register ........................................ 3-65, 3-359, 3-373, 3-384, 3-458, 4-83
CVTDQ2PD instruction ............................... 3-142
CVTDQ2PS instruction ............................... 3-144
Vol. 2B INDEX-1
INDEX

CVTPD2DQ instruction ........................................... 3-146
CVTPD2PI instruction ........................................... 3-148
CVTPD2PS instruction ........................................... 3-150
CVTP2PD instruction ........................................... 3-152
CVTP2PS instruction ........................................... 3-154
CVTPS2DQ instruction ........................................... 3-156
CVTPS2PD instruction ........................................... 3-158
CVTPS2PI instruction ........................................... 3-160
CVTSD2SI instruction ........................................... 3-162
CVTSD2SS instruction ........................................... 3-164
CVTSS2SD instruction ........................................... 3-166
CVTSS2SI instruction ........................................... 3-168
CVTSS2SD instruction ........................................... 3-170
CVTSS2SI instruction ........................................... 3-172
CVTTPD2DQ instruction ........................................ 3-176
CVTTPD2PI instruction ........................................ 3-174
CVTTPS2DQ instruction ........................................ 3-178
CVTTPS2PI instruction ........................................ 3-180
CVTTSD2SI instruction ........................................ 3-182
CVTTSD2SI instruction ........................................ 3-184
CWD instruction ................................................. 3-186
CWEDE instruction .............................................. 3-76
C/C++ compiler intrinsics
  compiler functional equivalents .......................... C-1
  composite ...................................................... C-32
  description of .............................................. 3-9
  lists of ...................................................... C-1
  simple ........................................................ C-3

D
D (default operation size) flag, segment descriptor ........ 4-83, 4-88, 4-141
DAA instruction ................................................ 3-188
DAS instruction ................................................ 3-190
Debug registers, moving value to and from .................. 3-464
DEC instruction ................................................. 3-192, 3-416
Denormalized finite number ................................... 3-311
DF (direction) flag, EFLAGS register ........................ 3-79, 3-99, 3-356, 3-418, 3-512, 4-14, 4-186, 4-219
Displacement (operand addressing) ............................ 2-4
DIV instruction ................................................ 3-194
Divde error exception (#DE) .................................. 3-194
DIVPD instruction ............................................. 3-197
DIVPS instruction ............................................. 3-199
DIVSD instruction ............................................. 3-201
DIVSS instruction ............................................. 3-203
DS register ...................................................... 3-99, 3-399, 3-418, 3-512, 4-14

E
EDI register ..................................................... 4-186, 4-219, 4-225
Effective address ............................................... 3-402
EFLAGS register
  condition codes ............................................. 3-87, 3-223, 3-228
  flags affected by instructions ............................. 3-11
  loading ....................................................... 3-391
  popping ....................................................... 4-90
  popping on return from interrupt ........................ 3-373
  pushing ...................................................... 4-146
  pushing on interrupts ...................................... 3-359
  saving ....................................................... 4-179
  status flags ................................................ 3-89, 3-381, 4-189, 4-246
  EIP register ................................................ 3-65, 3-359, 3-373, 3-384
  EMMS instruction ........................................... 3-205
  Encoding
    cacheability and memory ordering
      instructions ............................................. B-32
      cacheability instructions ................................ B-45
    SIMD integer register field .............................. B-31, B-40
  ENTER instruction .......................................... 3-206
  ES register ................................................ 3-399, 4-14, 4-186, 4-225
  ESI register ................................................. 3-99, 3-418, 3-512, 4-14, 4-219
  ESP register ................................................ 3-66, 4-84
  Exceptions
    BOUND range exceeded (#BR) ................................ 3-50
    notation ................................................... 1-5
    overflow exception (#OF) .................................. 3-359
    returning from ........................................... 3-373
  Exponent, extracting from floating-point number .......... 3-325
  Extract exponent and significand, x87 FPU operation .......... 3-325

F
F2XM1 instruction ............................................. 3-209, 3-325
FABS instruction ............................................. 3-211
FADD instruction ............................................. 3-212
FADDP instruction ............................................ 3-212
Far call, CALL instruction .................................... 3-65
Far pointer, loading .......................................... 3-399
Far return, RET instruction .................................. 4-167
FBLD instruction ............................................. 3-215
FBSTP instruction ............................................. 3-217
FCHS instruction .............................................. 3-220
FCLX/FNCLEX instructions .................................... 3-221
FCMOVcc instructions ......................................... 3-223
FCOM instruction ............................................. 3-225
FCOMI instruction ............................................ 3-228
FCOMIP instruction ............................................ 3-228
FCOMPP instruction ............................................ 3-225
FCOS instruction .............................................. 3-231
FDECSTFP instruction .......................................... 3-233
FDIV instruction .............................................. 3-234
FDIV instruction .............................................. 3-234
FDIV instruction .............................................. 3-237
FDIVR instruction .............................................. 3-237
FDIVR instruction .............................................. 3-237
Feature information, processor ................................ 3-120
FFREE instruction .............................................. 3-240
FIADD instruction ............................................. 3-212
FICOMP instruction ............................................ 3-241
FICOMP instruction ............................................ 3-241
FIDIV instruction .............................................. 3-234
FIDIV instruction .............................................. 3-237

INDEX-2 Vol. 2B
INDEX

Instruction set, reference .................................. 3-1
INSW instruction ........................................... 3-356
INT 3 instruction .......................................... 3-359
Integer, storing, x87 FPU data type .................... 3-248
Intel Xeon processor ........................................ 1-1
Inter-privilege level
   call, CALL instruction ................................ 3-65
   return, RET instruction ................................ 4-167
Interrupts
   interrupt vector 4 ...................................... 3-359
   returning from software ................................ 3-373
INTn instruction ........................................... 3-359
INTO instruction .......................................... 3-359
Intrinsics
   compiler functional equivalents ...................... C-1
   composite ............................................. C-32
   description of ....................................... 3-9
   list of ................................................ C-1
   simple .................................................. C-3
INVD instruction .......................................... 3-370
INVLPG instruction ........................................ 3-372
IOPL (I/O privilege level) field, EFLAGS register .... 3-82, 4-146, 4-220
IRET instruction .......................................... 3-373
IRETD instruction ......................................... 3-373

J
Jcc instructions ............................................. 3-380
JMP instruction ............................................ 3-384
Jump operation ............................................. 3-384

L
LAHF instruction ............................................ 3-391
LAR instruction ............................................ 3-392
LDDQQU instruction ....................................... 3-395
LDMXCSR instruction ...................................... 3-397
LDS instruction ............................................. 3-399
LDT (local descriptor table) .................................. 3-411
LDTR (local descriptor table register) .................. 3-411, 4-206
LEA instruction ............................................. 3-402
LEAVE instruction ......................................... 3-404
LES instruction ............................................. 3-399
LFENCE instruction ........................................ 3-407
LFS instruction ............................................. 3-399
LGDT instruction ............................................ 3-409
LGS instruction ............................................. 3-399
LIDT instruction ............................................ 3-409
LLDT instruction ............................................ 3-411
LMSW instruction ............................................ 3-414
Load effective address operation ........................ 3-402
LOCK prefix3-20, 3-22, 3-38, 3-59, 3-61, 3-63, 3-110, 3-112, 3-192, 3-354, 3-416, 4-1, 4-4, 4-6, 4-184, 4-229, 4-270, 4-272, 4-276

Locking operation .......................................... 3-416
LODS instruction .......................................... 3-418, 4-164
LODSB instruction .......................................... 3-418
LODDSD instruction ......................................... 3-418
LODSW instruction .......................................... 3-418
Log epsilon, x87 FPU operation .......................... 3-327
Log (base 2), x87 FPU operation .......................... 3-329
LOOP instructions ......................................... 3-421
LOOPcc instructions ....................................... 3-421
LSL instruction ............................................. 3-423
LSS instruction ............................................. 3-399
LTR instruction ............................................. 3-427

M
Machine status word, CR0 register ....................... 3-414, 4-208
MASKMOVDQU instruction .................................. 3-429
MASKMOVQ instruction ..................................... 3-431
MAXPD instruction ......................................... 3-434
MAXPS instruction ......................................... 3-437
MAXSD instruction ......................................... 3-440
MAXSS instruction ......................................... 3-442
MFENCE instruction ........................................ 3-444
MINPD instruction ......................................... 3-445
MINPS instruction ......................................... 3-448
MINS instruction .......................................... 3-451
MINS instruction .......................................... 3-453
Mod field, instruction format ................................ 2-4
ModR/M byte ................................................. 2-4
16-bit addressing forms .................................... 2-6
32-bit addressing forms .................................... 2-7
description of ............................................. 2-4
format of .................................................. 2-1
MONITOR instruction ....................................... 3-455
CPUID flag ................................................... 3-127
MOV instruction ............................................. 3-458
MOV instruction (control registers) ....................... 3-462
MOV instruction (debug registers) ......................... 3-464
MOVAPD instruction ....................................... 3-466
MOVAPS instruction ....................................... 3-468
MOV instruction ............................................. 3-470
MOVDUP instruction ....................................... 3-473
MOVQ2Q instruction ....................................... 3-480
MOVDQA instruction ....................................... 3-473
MOVDQ instruction ......................................... 3-473
MOVHLPS instruction ...................................... 3-481
MOVHPPD instruction ...................................... 3-482
MOVHPS instruction ....................................... 3-484
MOVLHPS instruction ...................................... 3-486
MOVLPD instruction ....................................... 3-487
MOVLP instruction ......................................... 3-489
MOVMSKPD instruction .................................... 3-491
MOVMSKPS instruction .................................... 3-492
MOVTQDQ instruction ...................................... 3-493
MOVNTI instruction ........................................ 3-495
MOVNTQDQ instruction .................................... 3-497
MOVTQTPD instruction .................................... 3-497
MOVNTPS instruction ...................................... 3-499
INDEX

MOVNTQ instruction . 3-501
MOVQ instruction . 3-509
MOVG2DQ instruction . 3-511
MOVS instruction . 3-512, 4-164
MOVSX instruction . 3-519
MOVD instruction . 3-512, 3-515
MOVSHEUD instruction . 3-503
MOVSQ instruction . 3-506
Move instruction . 3-517
MOSW instruction . 3-512
MOVSQ instruction . 3-519
MOVUPD instruction . 3-520
MOVUPS instruction . 3-522
MOVZX instruction . 3-524
MSRs (model specific registers)
  reading . 4-158
  writing . 4-268
MUL instruction . 3-18, 3-525
MULP instruction . 3-527
MULPS instruction . 3-529
MULSD instruction . 3-531
MULSS instruction . 3-533
MVI instruction . 3-535
CPUID flag . 3-127

N
Na, testing for . 3-305
Near
  call, CALL instruction . 3-65
  return, RET instruction . 4-167
NEG instruction . 3-416, 4-1
NetBurst microarchitecture (see Intel NetBurst microarchitecture)
  Nomenclature, used in instruction reference pages . 3-1
  Nonconforming code segment . 3-387
  NOP instruction . 4-3
  NOT instruction . 3-416, 4-4
  Notation
    bit and byte order . 1-2
    exceptions . 1-5
    hexadecimal and binary numbers . 1-4
    instruction operands . 1-4
    reserved bits . 1-3
    reserved opcodes . 2-2
    segmented addressing . 1-5
  Notational conventions
  NT (nested task) flag, EFLAGS register . 3-373

O
OF (carry) flag, EFLAGS register . 3-350
OF (overflow) flag, EFLAGS register . 3-20, 3-22, 3-359, 3-525, 4-184, 4-195, 4-197, 4-229
Op code
  escape instructions . A-14
  mapping . A-1
  Opcode extensions
    description . A-13
    table . A-13
  Opcode format . 2-3
  Opcode integer instructions
    one-byte . A-4
    one-byte opcode map . A-7, A-8
    two-byte . A-5
  Opcode key abbreviations . A-1
  Operand, instruction . 4-14
  OR instruction . 3-416, 4-6
  ORPD instruction . 4-8
  ORPS instruction . 4-10
  OUT instruction . 4-12
  OUTS instruction . 4-14, 4-164
  OUTSB instruction . 4-14
  OUTSD instruction . 4-14
  OUTSW instruction . 4-14
  Overflow exception (#OF) . 3-359

P
P6 family processors
  description of . 1-1
  PACKSSDW instruction . 4-17
  PACKSSWB instruction . 4-17
  PACKUSWB instruction . 4-21
  PADDQ instruction . 4-27
  PADDDB instruction . 4-29
  PADDSSW instruction . 4-29
  PADDSW instruction . 4-32
  PADDSUBW instruction . 4-32
  PAND instruction . 4-35
  PANDN instruction . 4-37
  PAUSE instruction . 4-39
  PAVGB instruction . 4-40
  PAVGW instruction . 4-40
  PCESW instruction . 4-40
  PCE flag, CR4 register . 4-159
  PCMPEQB instruction . 4-43
  PCMPEQD instruction . 4-43
  PCMPEQd instruction . 4-43
  PCMPGTB instruction . 4-47
  PCMPGTQ instruction . 4-47
  PCMPGTW instruction . 4-47
  PE (protection-able) flag, CR0 register . 3-414
  Pentium 4 processor . 1-1
  Pentium processor . 1-1
  Pentium III processor . 1-1
  Pentium Pro processor . 1-1
  Pentium M processor . 1-1
  Pentium M processor . 1-1
  Performance-monitoring counters
  reading . 4-159
  PEXTRW instruction . 4-51
  Pi
  loading . 3-255
  PINSRW instruction . 4-53

Vol. 2B INDEX-5
INDEX

SHL instruction ........................................ 4-180
SHLD instruction ....................................... 4-195
SHR instruction ......................................... 4-180
SHRD instruction ........................................ 4-197
SHUFPD instruction ..................................... 4-199
SHUFPS instruction .................................... 4-201
SIB byte .................................................. 2-4
32-bit addressing forms of ................................ 2-8
description of ........................................... 2-4
format of ................................................. 2-1
SIDT instruction .......................................... 4-204
Significand, extracting from floating-point number ........................................... 3-325
SIMD floating-point exceptions, u nmasking, flags of ........................................... 3-397
Sine, x87 FPU operation ................................ 3-283, 3-285
SLDT instruction .......................................... 4-206
SMSW instruction ......................................... 4-208
SQRTPD instruction ....................................... 4-210
SQRTPS instruction ....................................... 4-212
SQRTSD instruction ....................................... 4-214
SQRTSS instruction ....................................... 4-216
Square root, Fx87 PU operation .......................... 3-287
SS register ................................................ 3-399, 3-459, 4-84
SSE extensions
  encoding cacheability and memory ordering instructions ........................................... B-32
  encoding SIMD-integer register field ................................................................. B-31
SSE2 extensions
  encoding cacheability instructions .......................... B-45
coding SIMD-integer register field ............................................................... B-40
SSE3 extensions
  CPUID extended function information ............................................................... 3-124
  CPUID flag ............................................... 3-127
  formats and encoding tables ................................................................. B-46
Stack, pushing values on .................................. 3-141
Status flags, EFLAGS register ............................ 3-87, 3-89, 3-223, 3-228, 3-381, 4-189, 4-246
STC instruction ........................................... 4-218
STD instruction ........................................... 4-219
STI instruction ........................................... 4-220
STMXCSR instruction ...................................... 4-223
STOSB instruction ......................................... 4-225
STOSD instruction ........................................ 4-225
STOSW instruction ......................................... 4-225
STR instruction ........................................... 4-228
String instructions ........................................ 3-99, 3-356, 3-418, 3-512, 4-14, 4-186, 4-225
SUB instruction ........................................... 3-19, 3-190, 3-416, 4-229
SUBPD instruction ......................................... 4-231
SUBSS instruction ......................................... 4-237
SYSEXENTER instruction .................................. 4-239
SYSEXIT instruction ....................................... 4-243
T
Tangent, x87 FPU operation ................................ 3-273
Task gate .................................................... 3-388
Task register
  loading .................................................. 3-427
  storing .................................................. 4-228
Task switch
  CALL instruction ........................................ 3-65
  return from nested task, IRET instruction ..................................................... 3-373
TEST instruction ......................................... 4-246
Time-stamp counter, reading ............................. 4-162
TLB entry, invalidating (flushing) ....................... 3-372
TS (task switched) flag, CR0 register ................... 3-84
TSD flag, CR4 register .................................... 4-162
TSS, relationship to task register ....................... 4-228
U
UCOMISD instruction ...................................... 4-248
UCOMISS instruction ...................................... 4-251
UD2 instruction ........................................... 4-254
Undefined, format opcodes ................................ 3-305
Unordered values ........................................ 3-225, 3-305, 3-307
UNPCKHPD instruction .................................... 4-255
UNPCKHPS instruction .................................... 4-257
UNPCKLPD instruction .................................... 4-259
UNPCKLPS instruction .................................... 4-261
V
VERR instruction ......................................... 4-263
Version information, processor .......................... 3-120
VERW instruction ......................................... 4-263
VM (virtual 8086 mode) flag, EFLAGS register .......... 3-373
W
WAIT/FWAIT instructions ................................ 4-265
WBINVD instruction ....................................... 4-266
Write-back and invalidate caches ........................ 4-266
WRMSR instruction ....................................... 4-268
X
x87 FPU
  checking for pending x87 FPU exceptions ................. 4-265
  constants ............................................... 3-255
  initialization .......................................... 3-246
x87 FPU control word
  loading .................................................. 3-257, 3-259
  RC field ................................................ 3-249, 3-255, 3-289
  restoring ............................................... 3-276

Vol. 2B INDEX-7
saving ........................................ 3-278, 3-294
storing ....................................... 3-292
x87 FPU data pointer ....................... 3-259, 3-276, 3-278, 3-294
x87 FPU instruction pointer ............... 3-259, 3-276, 3-278, 3-294
x87 FPU last opcode ....................... 3-259, 3-276, 3-278, 3-294
x87 FPU status word
  condition code flags 3-225, 3-241, 3-305, 3-307, 3-311
loading ...................................... 3-259
restoring .................................... 3-276
saving ....................................... 3-278, 3-294, 3-297
TOP field .................................... 3-245
x87 FPU flags affected by instructions .. 3-12
x87 FPU tag word ......................... 3-259, 3-276, 3-278, 3-294
XADD instruction ......................... 3-416, 4-270
XCHG instruction ......................... 3-416, 4-272
XLAT/XLATB instruction .................. 4-274
XOR instruction ............................ 3-416, 4-276
XORPD instruction ......................... 4-278
XORPS instruction ......................... 4-280

Z
ZF (zero) flag, EFLAGS register .......... 3-110, 3-112, 3-392, 3-421, 3-423, 4-164, 4-263