10

Programming With the Streaming SIMD Extensions (SSE)
CHAPTER 10
PROGRAMMING WITH
STREAMING SIMD EXTENSIONS (SSE)

The streaming SIMD extensions (SSE) were introduced into the IA-32 architecture in the Pentium III processor family. These extensions enhance the performance of IA-32 processors for advanced 2-D and 3-D graphics, motion video, image processing, speech recognition, audio synthesis, telephony, and video conferencing.

This chapter describes SSE extensions. Chapter 11, Programming with Streaming SIMD Extensions 2 (SSE2) provides information to assist in writing application programs that use these extensions and the SSE2 extensions. Chapter 12, Programming with Streaming SIMD Extensions 3 (SSE3) provides information for the SSE3 extensions.

10.1. OVERVIEW OF SSE EXTENSIONS

Intel MMX technology introduced single-instruction multiple-data (SIMD) capability into the IA-32 architecture, with the 64-bit MMX registers, 64-bit packed integer data types, and instructions that allowed SIMD operations to be performed on packed integers. SSE extensions expand the SIMD execution model by adding facilities for handling packed and scalar single-precision floating-point values contained in 128-bit registers.

SSE extensions add the following features to the IA-32 architecture, while maintaining backward compatibility with all existing IA-32 processors, applications and operating systems.

- Eight 128-bit data registers, called XMM registers
- The 32-bit MXCSR register, which provides control and status bits for operations performed on XMM registers
- The 128-bit packed single-precision floating-point data type (four IEEE single-precision floating-point values packed into a double quadword)
- Instructions that perform SIMD operations on single-precision floating-point values and that extend SIMD operations that can be performed on integers:
  - 128-bit Packed and scalar single-precision floating-point instructions that operate on data located in MMX registers
  - 64-bit SIMD integer instructions that support additional operations on packed integer operands located in MMX registers
- Instructions that save and restore the state of the MXCSR register
PROGRAMMING WITH STREAMING SIMD EXTENSIONS (SSE)

- Instructions that support explicit prefetching of data, control of the cacheability of data, and control the ordering of store operations
- Extensions to the CPUID instruction

These features extend the IA-32 architecture’s SIMD programming model in four important ways:

- The ability to perform SIMD operations on four packed single-precision floating-point values enhances the performance of IA-32 processors for advanced media and communications applications that use computation-intensive algorithms to perform repetitive operations on large arrays of simple, native data elements.
- The ability to perform SIMD single-precision floating-point operations in XMM registers and SIMD integer operations in MMX registers provides greater flexibility and throughput for executing applications that operate on large arrays of floating-point and integer data.
- Cache control instructions provide the ability to stream data in and out of XMM registers without polluting the caches and the ability to prefetch data to selected cache levels before it is actually used. Applications that require regular access to large amounts of data benefit from these prefetching and streaming store capabilities.
- The SFENCE (store fence) instruction provides greater control over the ordering of store operations when using weakly-ordered memory types.

SSE extensions are fully compatible with all software written for IA-32 processors. All existing software continues to run correctly, without modification, on processors that incorporate SSE extensions. Enhancements to CPUID permit detection of SSE extensions. SSE extensions are accessible from all IA-32 execution modes: protected mode, real address mode, and virtual-8086 mode.

The following sections of this chapter describe the programming environment for SSE extensions, including: XMM registers, the packed single-precision floating-point data type, and SSE instructions. For additional information, see:

- Section 11.6., “Writing Applications with the SSE and SSE2 Extensions”.
- Section 11.5., “SSE, SSE2, and SSE3 Exceptions”, describes the exceptions that can be generated with SSE/SSE2/SSE3 instructions.
- Chapter 12, SSE and SSE2 System Programming, in the IA-32 Intel Architecture Software Developer’s Manual, Volume 3 gives guidelines for integrating these extensions into an operating-system environment.
10.2. SSE PROGRAMMING ENVIRONMENT

Figure 10-1 shows the execution environment for the SSE extensions. All SSE instructions operate on the XMM registers, MMX registers, and/or memory as follows:

- **XMM registers.** These eight registers (see Figure 10-2 and Section 10.2.1., “XMM Registers”) are used to operate on packed or scalar single-precision floating-point data. Scalar operations are operations performed on individual (unpacked) single-precision floating-point values stored in the low doubleword of an XMM register. XMM registers are referenced by the names XMM0 through XMM7.

- **MXCSR register.** This 32-bit register (see Figure 10-3 and Section 10.2.2., “MXCSR Control and Status Register”) provides status and control bits used in SIMD floating-point operations.

- **MMX registers.** These eight registers (see Figure 9-2) are used to perform operations on 64-bit packed integer data. They are also used to hold operands for some operations performed between the MMX and XMM registers. MMX registers are referenced by the names MM0 through MM7.

- **General-purpose registers.** The eight general-purpose registers (see Figure 3-4) are used along with the existing IA-32 addressing modes to address operands in memory. (MMX and XMM registers cannot be used to address memory). The general-purpose registers are also used to hold operands for some SSE instructions and are referenced as EAX, EBX, ECX, EDX, EBP, ESI, EDI, and ESP.
10-4 Vol. 1

PROGRAMMING WITH STREAMING SIMD EXTENSIONS (SSE)

- **EFLAGS register.** This 32-bit register (see Figure 3-7) is used to record result of some compare operations.

10.2.1. XMM Registers

Eight 128-bit XMM data registers were introduced into the IA-32 architecture with SSE extensions (see Figure 10-2). These registers can be accessed directly using the names XMM0 to XMM7; and they can be accessed independently from the x87 FPU and MMX registers and the general-purpose registers (that is, they are not aliased to any other of the processor’s registers).

![Figure 10-2. XMM Registers](image)

SSE instructions use the XMM registers only to operate on packed single-precision floating-point operands. SSE2 extensions expand the functions of the XMM registers to operand on packed or scalar double-precision floating-point operands and packed integer operands (see Section 11.2., “SSE2 Programming Environment” and Section 12.2., “SSE3 Programming Environment and Data types”).

XMM registers can only be used to perform calculations on data; they cannot be used to address memory. Addressing memory is accomplished by using the general-purpose registers.

Data can be loaded into XMM registers or written from the registers to memory in 32-bit, 64-bit, and 128-bit increments. When storing the entire contents of an XMM register in memory (128-bit store), the data is stored in 16 consecutive bytes, with the low-order byte of the register being stored in the first byte in memory.
10.2.2. MXCSR Control and Status Register

The 32-bit MXCSR register (see Figure 10-3) contains control and status information for SSE, SSE2 and SSE3 SIMD floating-point operations. This register contains:

- flag and mask bits for SIMD floating-point exceptions
- rounding control field for SIMD floating-point operations
- flush-to-zero flag that provides a means of controlling underflow conditions on SIMD floating-point operations
- denormals-are-zeros flag that controls how SIMD floating-point instructions handle denormal source operands

The contents of this register can be loaded from memory with the LDMXCSR and FXRSTOR instructions and stored in memory with STMXCSR and FXSAVE.

Bits 16 through 31 of the MXCSR register are reserved and are cleared on a power-up or reset of the processor; attempting to write a non-zero value to these bits, using either the FXRSTOR or LDMXCSR instructions, will result in a general-protection exception (#GP) being generated.

* The denormals-are-zeros flag was introduced in the Pentium 4 and Intel Xeon processor.
10.2.2.1. SIMD Floating-Point Mask and Flag Bits

Bits 0 through 5 of the MXCSR register indicate whether an SIMD floating-point exception has been detected. They are “sticky” flags. That is, after a flag is set, it remains set until explicitly cleared. To clear these flags, use the LDMXCSR or the FXRSTOR instruction to write zeroes to them.

Bits 7 through 12 provide individual mask bits for the SIMD floating-point exceptions. An exception type is masked if the corresponding mask bit is set, and it is unmasked if the bit is clear. These mask bits are set upon a power-up or reset. This causes all SIMD floating-point exceptions to be initially masked.

If LDMXCSR or FXRSTOR clears a mask bit and sets the corresponding exception flag bit, an SIMD floating-point exception will not be generated as a result of this change. The unmasked exception will be generated only upon the execution of the next SSE/SSE2/SSE3 instruction that detects the unmasked exception condition.

For more information about the use of the SIMD floating-point exception mask and flag bits, see Section 11.5., “SSE, SSE2, and SSE3 Exceptions” and Section 12.4., “SSE3 Exceptions”.

10.2.2.2. SIMD Floating-Point Rounding Control Field

Bits 13 and 14 of the MXCSR register (the rounding control [RC] field) control how the results of SIMD floating-point instructions are rounded. See Section 4.8.4., “Rounding” for a description of the function and encoding of the rounding control bits.

10.2.2.3. Flush-To-Zero

Bit 15 (FZ) of the MXCSR register enables the flush-to-zero mode, which controls the masked response to an SIMD floating-point underflow condition. When the underflow exception is masked and the flush-to-zero mode is enabled, the processor performs the following operations when it detects a floating-point underflow condition:

• Returns a zero result with the sign of the true result
• Sets the precision and underflow exception flags

If the underflow exception is not masked, the flush-to-zero bit is ignored.

The flush-to-zero mode is not compatible with IEEE Standard 754. The IEEE-mandated masked response to underflow is to deliver the denormalized result (see Section 4.8.3.2., “Normalized and Denormalized Finite Numbers”). The flush-to-zero mode is provided primarily for performance reasons. At the cost of a slight precision loss, faster execution can be achieved for applications where underflows are common and rounding the underflow result to zero can be tolerated.

The flush-to-zero bit is cleared upon a power-up or reset of the processor, disabling the flush-to-zero mode.
10.2.2.4. Denormals-Are-Zeros

Bit 6 (DAZ) of the MXCSR register enables the denormals-are-zeros mode, which controls the processor’s response to an SIMD floating-point denormal operand condition. When the denormals-are-zeros flag is set, the processor converts all denormal source operands to a zero with the sign of the original operand before performing any computations on them. The processor does not set the denormal-operand exception flag (DE), regardless of the setting of the denormal-operand exception mask bit (DM); and it does not generate a denormal-operand exception if the exception is unmasked.

The denormals-are-zeros mode is not compatible with IEEE Standard 754 (see Section 4.8.3.2., “Normalized and Denormalized Finite Numbers”). The denormals-are-zeros mode is provided to improve processor performance for applications such as streaming media processing, where rounding a denormal operand to zero does not appreciably affect the quality of the processed data.

The denormals-are-zeros flag is cleared upon a power-up or reset of the processor, disabling the denormals-are-zeros mode.

The denormals-are-zeros mode was introduced in the Pentium 4 and Intel Xeon processor with the SSE2 extensions; however, it is fully compatible with the SSE SIMD floating-point instructions (that is, the denormals-are-zeros flag affects the operation of the SSE SIMD floating-point instructions). In earlier IA-32 processors and in some models of the Pentium 4 processor, this flag (bit 6) is reserved. See Section 11.6.3., “Checking for the DAZ Flag in the MXCSR Register” for instructions for detecting the availability of this feature.

Attempting to set bit 6 of the MXCSR register on processors that do not support the DAZ flag will cause a general-protection exception (#GP). See Section 11.6.6., “Guidelines for Writing to the MXCSR Register” for instructions for preventing such general-protection exceptions by using the MXCSR_MASK value returned by the FXSAVE instruction.

10.2.3. Compatibility of the SSE Extensions with SSE2 and SSE3 Extensions, MMX Technology, and the x87 FPU Programming Environments

The state (XMM registers and MXCSR register) introduced into the IA-32 execution environment with the SSE extensions is shared with SSE2 and SSE3 extensions. SSE/SSE2/SSE3 instructions are fully compatible; they can be executed together in the same instruction stream with no need to save state when switching between instruction sets.

XMM registers are independent of the x87 FPU and MMX registers, so SSE/SSE2/SSE3 operations performed on the XMM registers can be performed in parallel with operations on the x87 FPU and MMX registers (see Section 11.6.7., “Interaction of SSE and SSE2 Instructions with x87 FPU and MMX Instructions”).

The FXSAVE and FXRSTOR instructions save and restore the SSE/SSE2/SSE3 states along with the x87 FPU and MMX state.
10.3. SSE DATA TYPES

SSE extensions introduced one data type, the 128-bit packed single-precision floating-point data type, to the IA-32 architecture (see Figure 10-4). This data type consists of four IEEE 32-bit single-precision floating-point values packed into a double quadword. (See Figure 4-3 for the layout of a single-precision floating-point value; refer to Section 4.2.2., “Floating-Point Data Types” for a detailed description of the single-precision floating-point format.)

This 128-bit packed single-precision floating-point data type is operated on in the XMM registers or in memory. Conversion instructions are provided to convert two packed single-precision floating-point values into two packed doubleword integers or a scalar single-precision floating-point value into a doubleword integer (see Figure 11-8).

SSE extensions provide conversion instructions between XMM registers and MMX registers, and between XMM registers and general-purpose bit registers. See Figure 11-8.

The address of a 128-bit packed memory operand must be aligned on a 16-byte boundary, except in the following cases:

• The MOVUPS instruction supports unaligned accesses.
• Scalar instructions that use a 4-byte memory operand that is not subject to alignment requirements.

Figure 4-2 shows the byte order of 128-bit (double quadword) data types in memory.

10.4. SSE INSTRUCTION SET

SSE instructions are divided into four functional groups

• Packed and scalar single-precision floating-point instructions
• 64-bit SIMD integer instructions
• State management instructions
• Cacheability control, prefetch, and memory ordering instructions

The SSE CPUID feature bit (bit 25 in the EDX register) indicates whether the SSE instructions have been implemented in an IA-32 processor implementation.

The following sections give an overview of each of the instructions in these groups.
10.4.1. SSE Packed and Scalar Floating-Point Instructions

The packed and scalar single-precision floating-point instructions are divided into the following subgroups:

- Data movement instructions
- Arithmetic instructions
- Logical instructions
- Comparison instructions
- Shuffle instructions
- Conversion instructions

The packed single-precision floating-point instructions perform SIMD operations on packed single-precision floating-point operands (see Figure 10-5). Each source operand contains four single-precision floating-point values, and the destination operand contains the results of the operation (OP) performed in parallel on the corresponding values (X0 and Y0, X1 and Y1, X2 and Y2, and X3 and Y3) in each operand.

![Figure 10-5. Packed Single-Precision Floating-Point Operation](image)

The scalar single-precision floating-point instructions operate on the low (least significant) doublewords of the two source operands (X0 and Y0); see Figure 10-6. The three most significant doublewords (X1, X2, and X3) of the first source operand are passed through to the destination. The scalar operations are similar to the floating-point operations performed in the x87 FPU data registers with the precision control field in the x87 FPU control word set for single precision (24-bit significand), except that x87 stack operations use a 15-bit exponent range for the result, while SSE operations use an 8-bit exponent range.
10.4.1.1. SSE Data Movement Instructions

SSE data movement instructions move single-precision floating-point data between XMM registers and between an XMM register and memory.

The MOVAPS (move aligned packed single-precision floating-point values) instruction transfers a double quadword operand containing four packed single-precision floating-point values from memory to an XMM register and vice versa, or between XMM registers. The memory address must be aligned to a 16-byte boundary; otherwise, a general-protection exception (#GP) is generated.

The MOVUPS (move unaligned packed single-precision, floating-point) instruction performs the same operations as the MOVAPS instruction, except that 16-byte alignment of a memory address is not required.

The MOVSS (move scalar single-precision floating-point) instruction transfers a 32-bit single-precision floating-point operand from memory to the low doubleword of an XMM register and vice versa, or between XMM registers.

The MOVLPS (move low packed single-precision floating-point) instruction moves two packed single-precision floating-point values from memory to the low quadword of an XMM register and vice versa. The high quadword of the register is left unchanged.

The MOVHPS (move high packed single-precision floating-point) instruction moves two packed single-precision floating-point values from memory to the high quadword of an XMM register and vice versa. The low quadword of the register is left unchanged.

The MOVLHPS (move packed single-precision floating-point low to high) instruction moves two packed single-precision floating-point values from the low quadword of the source XMM register into the high quadword of the destination XMM register. The low quadword of the destination register is left unchanged.

The MOVHLPS (move packed single-precision floating-point high to low) instruction moves two packed single-precision floating-point values from the high quadword of the source XMM
register into the low quadword of the destination XMM register. The high quadword of the destination register is left unchanged.

The MOVMSKPS (move packed single-precision floating-point mask) instruction transfers the most significant bit of each of the four packed single-precision floating-point numbers in an XMM register to a general-purpose register. This 4-bit value can then be used as a condition to perform branching.

10.4.1.2. SSE Arithmetic Instructions

SSE arithmetic instructions perform addition, subtraction, multiply, divide, reciprocal, square root, reciprocal of square root, and maximum/minimum operations on packed and scalar single-precision floating-point values.

The ADDPS (add packed single-precision floating-point values) and SUBPS (subtract packed single-precision floating-point values) instructions add and subtract, respectively, two packed single-precision floating-point operands.

The ADDSS (add scalar single-precision floating-point values) and SUBSS (subtract scalar single-precision floating-point values) instructions add and subtract, respectively, the low single-precision floating-point values of two operands and store the result in the low doubleword of the destination operand.

The MULPS (multiply packed single-precision floating-point values) instruction multiplies two packed single-precision floating-point operands.

The MULSS (multiply scalar single-precision floating-point values) instruction multiplies the low single-precision floating-point values of two operands and stores the result in the low doubleword of the destination operand.

The DIVPS (divide packed, single-precision floating-point values) instruction divides two packed single-precision floating-point operands.

The DIVSS (divide scalar single-precision floating-point values) instruction divides the low single-precision floating-point values of two operands and stores the result in the low doubleword of the destination operand.

The RCPPS (compute reciprocals of packed single-precision floating-point values) instruction computes the approximate reciprocals of values in a packed single-precision floating-point operand.

The RCPSS (compute reciprocal of scalar single-precision floating-point values) instruction computes the approximate reciprocal of the low single-precision floating-point value in the source operand and stores the result in the low doubleword of the destination operand.

The SQRTPS (compute square roots of packed single-precision floating-point values) instruction computes the square roots of the values in a packed single-precision floating-point operand.

The SQRTSS (compute square root of scalar single-precision floating-point values) instruction computes the square root of the low single-precision floating-point value in the source operand and stores the result in the low doubleword of the destination operand.
The RSQRTPS (compute reciprocals of square roots of packed single-precision floating-point values) instruction computes the approximate reciprocals of the square roots of the values in a packed single-precision floating-point operand.

The RSQRTSS (reciprocal of square root of scalar single-precision floating-point value) instruction computes the approximate reciprocal of the square root of the low single-precision floating-point value in the source operand and stores the result in the low doubleword of the destination operand.

The MAXPS (return maximum of packed single-precision floating-point values) instruction compares the corresponding values from two packed single-precision floating-point operands and returns the numerically greater value from each comparison to the destination operand.

The MAXSS (return maximum of scalar single-precision floating-point values) instruction compares the low values from two packed single-precision floating-point operands and returns the numerically greater value from the comparison to the low doubleword of the destination operand.

The MINPS (return minimum of packed single-precision floating-point values) instruction compares the corresponding values from two packed single-precision floating-point operands and returns the numerically lesser value from each comparison to the destination operand.

The MINSS (return minimum of scalar single-precision floating-point values) instruction compares the low values from two packed single-precision floating-point operands and returns the numerically lesser value from the comparison to the low doubleword of the destination operand.

### 10.4.2. SSE Logical Instructions

SSE logical instructions perform AND, AND NOT, OR, and XOR operations on packed single-precision floating-point values.

The ANDPS (bitwise logical AND of packed single-precision floating-point values) instruction returns the logical AND of two packed single-precision floating-point operands.

The ANDNPS (bitwise logical AND NOT of packed single-precision, floating-point values) instruction returns the logical AND NOT of two packed single-precision floating-point operands.

The ORPS (bitwise logical OR of packed single-precision, floating-point values) instruction returns the logical OR of two packed single-precision floating-point operands.

The XORPS (bitwise logical XOR of packed single-precision, floating-point values) instruction returns the logical XOR of two packed single-precision floating-point operands.
10.4.2.1. SSE Comparison Instructions

The compare instructions compare packed and scalar single-precision floating-point values and return the results of the comparison either to the destination operand or to the EFLAGS register.

The CMPPS (compare packed single-precision floating-point values) instruction compares the corresponding values from two packed single-precision floating-point operands, using an immediate operand as a predicate, and returns a 32-bit mask result of all 1s or all 0s for each comparison to the destination operand. The value of the immediate operand allows the selection of any of 8 compare conditions: equal, less than, less than equal, unordered, not equal, not less than, not less than or equal, or ordered.

The CMPSS (compare scalar single-precision, floating-point values) instruction compares the low values from two packed single-precision floating-point operands, using an immediate operand as a predicate, and returns a 32-bit mask result of all 1s or all 0s for the comparison to the low doubleword of the destination operand. The immediate operand selects the compare conditions as with the CMPPS instruction.

The COMISS (compare scalar single-precision floating-point values and set EFLAGS) and UCOMISS (unordered compare scalar single-precision floating-point values and set EFLAGS) instructions compare the low values of two packed single-precision floating-point operands and set the ZF, PF, and CF flags in the EFLAGS register to show the result (greater than, less than, equal, or unordered). These two instructions differ as follows: the COMISS instruction signals a floating-point invalid-operation (#I) exception when a source operand is either a QNaN or an SNaN; the UCOMISS instruction only signals an invalid-operation exception when a source operand is an SNaN.

10.4.2.2. SSE Shuffle and Unpack Instructions

SSE shuffle and unpack instructions shuffle or interleave the contents of two packed single-precision floating-point values and store the results in the destination operand.

The SHUFPS (shuffle packed single-precision floating-point values) instruction places any two of the four packed single-precision floating-point values from the destination operand into the two low-order doublewords of the destination operand, and places any two of the four packed single-precision floating-point values from the source operand in the two high-order doublewords of the destination operand (see Figure 10-7). By using the same register for the source and destination operands, the SHUFPS instruction can shuffle four single-precision floating-point values into any order.
The UNPCKHPS (unpack and interleave high packed single-precision floating-point values) instruction performs an interleaved unpack of the high-order single-precision floating-point values from the source and destination operands and stores the result in the destination operand (see Figure 10-8).

The UNPCKLPS (unpack and interleave low packed single-precision floating-point values) instruction performs an interleaved unpack of the low-order single-precision floating-point values from the source and destination operands and stores the result in the destination operand (see Figure 10-9).
10.4.3. SSE Conversion Instructions

SSE conversion instructions (see Figure 11-8) support packed and scalar conversions between single-precision floating-point and doubleword integer formats.

The CVTPI2PS (convert packed doubleword integers to packed single-precision floating-point values) instruction converts two packed signed doubleword integers into two packed single-precision floating-point values. When the conversion is inexact, the result is rounded according to the rounding mode selected in the MXCSR register.

The CVTSI2SS (convert doubleword integer to scalar single-precision floating-point value) instruction converts a signed doubleword integer into a single-precision floating-point value. When the conversion is inexact, the result is rounded according to the rounding mode selected in the MXCSR register.

The CVTPS2PI (convert packed single-precision floating-point values to packed doubleword integers) instruction converts two packed single-precision floating-point values into two packed signed doubleword integers. When the conversion is inexact, the result is rounded according to the rounding mode selected in the MXCSR register. The CVTTPS2PI (convert with truncation packed single-precision floating-point values to packed doubleword integers) instruction is similar to the CVTPS2PI instruction, except that truncation is used to round a source value to an integer value (see Section 4.8.4.2., “Truncation with SSE and SSE2 Conversion Instructions”).

The CVTSS2SI (convert scalar single-precision floating-point value to doubleword integer) instruction converts a single-precision floating-point value into a signed doubleword integer. When the conversion is inexact, the result is rounded according to the rounding mode selected in the MXCSR register. The CVTTSS2SI (convert with truncation scalar single-precision floating-point value to doubleword integer) instruction is similar to the CVTSS2SI instruction, except that truncation is used to round the source value to an integer value (see Section 4.8.4.2., “Truncation with SSE and SSE2 Conversion Instructions”).
10.4.4. SSE 64-bit SIMD Integer Instructions

SSE extensions add the following 64-bit packed integer instructions to the IA-32 architecture. These instructions operate on data in MMX registers and 64-bit memory locations.

**NOTE**
When SSE2 extensions are present in an IA-32 processor, these instructions are extended to operate on 128-bit operands in XMM registers and 128-bit memory locations.

The PAVGB (compute average of packed unsigned byte integers) and PAVGW (compute average of packed unsigned word integers) instructions compute an SIMD average of two packed unsigned byte or word integer operands, respectively. For each corresponding pair of data elements in the packed source operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position.

The PEXTRW (extract word) instruction copies a selected word from an MMX register into a general-purpose register.

The PINSRW (insert word) instruction copies a word from a general-purpose register or from memory into a selected word location in an MMX register.

The PMAXUB (maximum of packed unsigned byte integers) instruction compares the corresponding unsigned byte integers in two packed operands and returns the greater of each comparison to the destination operand.

The PMINUB (minimum of packed unsigned byte integers) instruction compares the corresponding unsigned byte integers in two packed operands and returns the lesser of each comparison to the destination operand.

The PMAXSW (maximum of packed signed word integers) instruction compares the corresponding signed word integers in two packed operands and returns the greater of each comparison to the destination operand.

The PMINSW (minimum of packed signed word integers) instruction compares the corresponding signed word integers in two packed operands and returns the lesser of each comparison to the destination operand.

The PMOVMSKB (move byte mask) instruction creates an 8-bit mask from the packed byte integers in an MMX register and stores the result in the low byte of a general-purpose register. The mask contains the most significant bit of each byte in the MMX register. (When operating on 128-bit operands, a 16-bit mask is created.)

The PMULHUW (multiply packed unsigned word integers and store high result) instruction performs an SIMD unsigned multiply of the words in the two source operands and returns the high word of each result to an MMX register.

The PSADBW (compute sum of absolute differences) instruction computes the SIMD absolute differences of the corresponding unsigned byte integers in two source operands, sums the differences, and stores the sum in the low word of the destination operand.
The PSHUFW (shuffle packed word integers) instruction shuffles the words in the source operand according to the order specified by an 8-bit immediate operand and returns the result to the destination operand.

10.4.5. MXCSR State Management Instructions

The MXCSR state management instructions (LDMXCSR and STMXCSR) load and save the state of the MXCSR register, respectively. The LDMXCSR instruction loads the MXCSR register from memory, while the STMXCSR instruction stores the contents of the register to memory.

10.4.6. Cacheability Control, Prefetch, and Memory Ordering Instructions

SSE extensions introduce several new instructions to give programs more control over the caching of data. They also introduces the PREFETCH instruction, which provide the ability to prefetch data to a specified cache level, and the SFENCE instruction, which enforces program ordering on stores. These instructions are described in the following sections.

10.4.6.1. Cacheability Control Instructions

The following three instructions enable data from the MMX and XMM registers to be stored to memory using a non-temporal hint. The non-temporal hint directs the processor to when possible store the data to memory without writing the data into the cache hierarchy. See Section 10.4.6.2., “Caching of Temporal vs. Non-Temporal Data” for information about non-temporal stores and hints.

- The MOVNTQ (store quadword using non-temporal hint) instruction stores packed integer data from an MMX register to memory, using a non-temporal hint.
- The MOVNTPS (store packed single-precision floating-point values using non-temporal hint) instruction stores packed floating-point data from an XMM register to memory, using a non-temporal hint.
- The MASKMOVQ (store selected bytes of quadword) instruction stores selected byte integers from an MMX register to memory, using a byte mask to selectively write the individual bytes. This instruction also uses a non-temporal hint.

10.4.6.2. Caching of Temporal vs. Non-Temporal Data

Data referenced by a program can be temporal (data will be used again) or non-temporal (data will be referenced once and not reused in the immediate future). For example, program code is generally temporal, whereas, multimedia data, such as the display list in a 3-D graphics application, is often non-temporal. To make efficient use of the processor’s caches, it is generally desirable to cache temporal data and not cache non-temporal data. Overloading the processor’s caches with non-temporal data is sometimes referred to as “polluting the caches.” The SSE and
SSE2 cacheability control instructions enable a program to write non-temporal data to memory in a manner that minimizes pollution of caches.

These SSE and SSE2 non-temporal store instructions minimize cache pollutions by treating the memory being accessed as the write combining (WC) type. If a program specifies a non-temporal store with one of these instructions and the destination region is mapped as cacheable memory (write back [WB], write through [WT] or WC memory type), the processor will do the following:

- If the memory location being written to is present in the cache hierarchy, the data in the caches is evicted.
- The non-temporal data is written to memory with WC semantics.

See also: Chapter 10, Memory Cache Control in the IA-32 Intel Architecture Software Developer’s Manual, Volume 3.

Using the WC semantics, the store transaction will be weakly ordered, meaning that the data may not be written to memory in program order, and the store will not write allocate (that is, the processor will not fetch the corresponding cache line into the cache hierarchy, prior to performing the store). Also, different processor implementations may choose to collapse and combine these stores.

The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in uncacheable memory. Uncacheable as referred to here means that the region being written to has been mapped with either an uncacheable (UC) or write protected (WP) memory type.

In general, WC semantics require software to ensure coherence, with respect to other processors and other system agents (such as graphics cards). Appropriate use of synchronization and fencing must be performed for producer-consumer usage models. Fencing ensures that all system agents have global visibility of the stored data; for instance, failure to fence may result in a written cache line staying within a processor and not being visible to other agents.

For processors that implement non-temporal stores by updating data in-place that already resides in the cache hierarchy, the destination region should also be mapped as WC. If mapped as WB or WT, there is the potential for speculative processor reads to bring the data into the caches; in this case, non-temporal stores would then update in place, and data would not be flushed from the processor by a subsequent fencing operation.

The memory type visible on the bus in the presence of memory type aliasing is implementation specific. As one possible example, the memory type written to the bus may reflect the memory type for the first store to this line, as seen in program order; other alternatives are possible. This behavior should be considered reserved, and dependence on the behavior of any particular implementation risks future incompatibility.

**10.4.6.3. PREFETCHh Instructions**

The PREFETCHh instructions permit programs to load data into the processor at a suggested cache level, so that the data is closer to the processor’s load and store unit when it is needed. These instructions fetch 32 aligned bytes (or more, depending on the implementation).
containing the addressed byte to a location in the cache hierarchy specified by the temporal locality hint (see Table 10-1). In this table, the first-level cache is closest to the processor and second-level cache is farther away from the processor than the first-level cache. The hints specify a prefetch of either temporal or non-temporal data (see Section 10.4.6.2., “Caching of Temporal vs. Non-Temporal Data”). Subsequent accesses to temporal data are treated like normal accesses, while those to non-temporal data will continue to minimize cache pollution. If the data is already present at a level of the cache hierarchy that is closer to the processor, the PREFETCH\(h\) instruction will not result in any data movement. The PREFETCH\(h\) instructions do not affect functional behavior of the program.

See Section 11.6.13., “Cacheability Hint Instructions” for additional information about the PREFETCH\(h\) instructions.

<table>
<thead>
<tr>
<th>PREFETCH(h) Instruction Mnemonic</th>
<th>Actions</th>
</tr>
</thead>
</table>
| PREFETCHT0                         | Temporal data—fetch data into all levels of cache hierarchy:  
  • Pentium III processor—1st-level cache or 2nd-level cache  
  • Pentium 4 and Intel Xeon processor—2nd-level cache |
| PREFETCHT1                         | Temporal data—fetch data into level 2 cache and higher  
  • Pentium III processor—2nd-level cache  
  • Pentium 4 and Intel Xeon processor—2nd-level cache |
| PREFETCHT2                         | Temporal data—fetch data into level 2 cache and higher  
  • Pentium III processor—2nd-level cache  
  • Pentium 4 and Intel Xeon processor—2nd-level cache |
| PREFETCHNTA                        | Non-temporal data—fetch data into location close to the processor, minimizing cache pollution  
  • Pentium III processor—1st-level cache  
  • Pentium 4 and Intel Xeon processor—2nd-level cache |

10.5. FXSAVE AND FXRSTOR INSTRUCTIONS

The FXSAVE and FXRSTOR instructions were introduced into the IA-32 architecture in the Pentium II processor family (prior to the introduction of the SSE extensions). The original versions of these instructions performed a fast save and restore, respectively, of the x87 FPU register state. (By saving the state of the x87 FPU data registers, the FXSAVE and FXRSTOR instructions implicitly save and restore the state of the MMX registers.)
The SSE extensions expanded the scope of these instructions to save and restore the states of the XMM registers and the MXCSR register, along with the x87 FPU and MMX state.

The FXSAVE and FXRSTOR instructions can be used in place of the FSAVE/FNSAVE and FRSTOR instructions; however, the operation of the FXSAVE and FXRSTOR instructions are not identical to the operation of FSAVE/FNSAVE and FRSTOR.

**NOTE**

The FXSAVE and FXRSTOR instructions are not considered part of the SSE instruction group. They have a separate CPUID feature bit (bit 24 of the EAX register) to indicate whether they are present on a particular IA-32 processor implementation. The CPUID feature bit for the SSE extensions (bit 25 of the EAX register) does not indicate the presence of the FXSAVE and FXRSTOR instructions.

10.6. HANDLING SSE INSTRUCTION EXCEPTIONS

See Section 11.5., “SSE, SSE2, and SSE3 Exceptions” for a detailed discussion of the general and SIMD floating-point exceptions that can be generated with the SSE instructions and for guidelines for handling these exceptions when they occur.

10.7. WRITING APPLICATIONS WITH THE SSE EXTENSIONS

See Section 11.6., “Writing Applications with the SSE and SSE2 Extensions” for additional information about writing applications and operating-system code using the SSE extensions.
11

Programming With the Streaming SIMD Extensions 2 (SSE2)
The streaming SIMD extensions 2 (SSE2) were introduced into the IA-32 architecture in the Pentium 4 and Intel Xeon processors. These extensions enhance the performance of IA-32 processors for advanced 3-D graphics, video decoding/encoding, speech recognition, E-commerce, Internet, scientific, and engineering applications.

This chapter describes the SSE2 extensions and provides information to assist in writing application programs that use these and the SSE extensions.

11.1. OVERVIEW OF SSE2 EXTENSIONS

SSE2 extensions use the single instruction multiple data (SIMD) execution model that is used with MMX technology and SSE extensions. They extend this model with support for packed double-precision floating-point values and for 128-bit packed integers.

SSE2 extensions add the following features to the IA-32 architecture, while maintaining backward compatibility with all existing IA-32 processors, applications and operating systems.

- Six data types:
  - 128-bit packed double-precision floating-point (two IEEE Standard 754 double-precision floating-point values packed into a double quadword)
  - 128-bit packed byte integers
  - 128-bit packed word integers
  - 128-bit packed doubleword integers
  - 128-bit packed quadword integers

- Instructions to support the additional data types and extend existing SIMD integer operations:
  - Packed and scalar double-precision floating-point instructions
  - Additional 64-bit and 128-bit SIMD integer instructions
  - 128-bit versions of SIMD integer instructions introduced with the MMX technology and the SSE extensions
  - Additional cacheability-control and instruction-ordering instructions
PROGRAMMING WITH STREAMING SIMD EXTENSIONS 2 (SSE2)

• Modifications to existing IA-32 instructions to support SSE2 features:
  — Extensions and modifications to the CPUID instruction
  — Modifications to the RDPMC instruction

These new features extend the IA-32 architecture’s SIMD programming model in three important ways:

• They provide the ability to perform SIMD operations on pairs of packed double-precision floating-point values. This permits higher precision computations to be carried out in XMM registers, which enhances processor performance in scientific and engineering applications and in applications that use advanced 3-D geometry techniques (such as ray tracing). Additional flexibility is provided with instructions that operate on single (scalar) double-precision floating-point values located in the low quadword of an XMM register.

• They provide the ability to operate on 128-bit packed integers (bytes, words, doublewords, and quadwords) in XMM registers. This provides greater flexibility and greater throughput when performing SIMD operations on packed integers. The capability is particularly useful for applications such as RSA authentication and RC5 encryption. Using the full set of SIMD registers, data types, and instructions provided with the MMX technology and SSE/SSE2 extensions, programmers can develop algorithms that finely mix packed single- and double-precision floating-point data and 64- and 128-bit packed integer data.

• SSE2 extensions enhance the support introduced with SSE extensions for controlling the cacheability of SIMD data. SSE2 cache control instructions provide the ability to stream data in and out of the XMM registers without polluting the caches and the ability to prefetch data before it is actually used.

SSE2 extensions are fully compatible with all software written for IA-32 processors. All existing software continues to run correctly, without modification, on processors that incorporate SSE2 extensions, as well as in the presence of applications that incorporate these extensions. Enhancements to the CPUID instruction permit detection of the SSE2 extensions. Also, because the SSE2 extensions use the same registers as the SSE extensions, no new operating-system support is required for saving and restoring program state during a context switch beyond that provided for the SSE extensions.

SSE2 extensions are accessible from all IA-32 execution modes: protected mode, real address mode, virtual 8086 mode.

The following sections in this chapter describe the programming environment for SSE2 extensions including: the 128-bit XMM floating-point register set, data types, and SSE2 instructions. It also describes exceptions that can be generated with the SSE and SSE2 instructions and gives guidelines for writing applications with SSE and SSE2 extensions.

For additional information about SSE2 extensions, see:

Chapter 12, SSE, SSE2 and SSE3 System Programming, in the IA-32 Intel Architecture Software Developer’s Manual, Volume 3 gives guidelines for integrating the SSE and SSE2 extensions into an operating-system environment.

11.2. SSE2 PROGRAMMING ENVIRONMENT

Figure 11-1 shows the programming environment for SSE2 extensions. No new registers or other instruction execution state are defined with SSE2 extensions. SSE2 instructions use the XMM registers, the MMX registers, and/or IA-32 general-purpose registers, as follows:

• **XMM registers.** These eight registers (see Figure 10-2) are used to operate on packed or scalar double-precision floating-point data. Scalar operations are operations performed on individual (unpacked) double-precision floating-point values stored in the low quadword of an XMM register. XMM registers are also used to perform operations on 128-bit packed integer data. They are referenced by the names XMM0 through XMM7.

• **MXCSR register.** This 32-bit register (see Figure 10-3) provides status and control bits used in floating-point operations. The denormals-are-zeros and flush-to-zero flags in this register provide a higher performance alternative for the handling of denormal source operands and denormal (underflow) results. For more information on the functions of these flags see Section 10.2.2.4., “Denormals-Are-Zeros” and Section 10.2.2.3., “Flush-To-Zero”.

---

**Figure 11-1. Steaming SIMD Extensions 2 Execution Environment**

- XMM Registers: Eight 128-Bit
- MXCSR Register: 32 Bits
- MMX Registers: Eight 64-Bit
- General-Purpose Registers: Eight 32-Bit
- EFLAGS Register: 32 Bits

---

Vol. 1 11-3
• **MMX registers.** These eight registers (see Figure 9-2) are used to perform operations on 64-bit packed integer data. They are also used to hold operands for some operations performed between MMX and XMM registers. MMX registers are referenced by the names MM0 through MM7.

• **General-purpose registers.** The eight general-purpose registers (see Figure 3-4) are used along with the existing IA-32 addressing modes to address operands in memory. MMX and XMM registers cannot be used to address memory. The general-purpose registers are also used to hold operands for some SSE2 instructions. These registers are referenced by the names EAX, EBX, ECX, EDX, EBP, ESI, EDI, and ESP.

• **EFLAGS register.** This 32-bit register (see Figure 3-7) is used to record the results of some compare operations.

### 11.2.1. Compatibility of SSE2 Extensions with SSE, MMX Technology, and x87 FPU Programming Environments

SSE2 extensions do not introduce any new state to the IA-32 execution environment beyond that of SSE. SSE2 extensions represent an enhancement of SSE extensions; they are fully compatible and share the same state information. SSE and SSE2 instructions can be executed together in the same instruction stream without the need to save state when switching between instruction sets.

XMM registers are independent of the x87 FPU and MMX registers; so SSE and SSE2 operations performed on XMM registers can be performed in parallel with x87 FPU or MMX technology operations (see Section 11.6.7., “Interaction of SSE and SSE2 Instructions with x87 FPU and MMX Instructions”).

The FXSAVE and FXRSTOR instructions save and restore the SSE and SSE2 states along with the x87 FPU and MMX states.

### 11.2.2. Denormals-Are-Zeros Flag

The denormals-are-zeros flag (bit 6 in the MXCSR register) was introduced into the IA-32 architecture with the SSE2 extensions. See Section 10.2.2.4., “Denormals-Are-Zeros” for a description of this flag.

### 11.3. SSE2 DATA TYPES

SSE2 extensions introduced one 128-bit packed floating-point data type and four 128-bit SIMD integer data types to the IA-32 architecture (see Figure 11-2).
• **Packed double-precision floating-point.** This 128-bit data type consists of two IEEE 64-bit double-precision floating-point values packed into a double quadword. (See Figure 4-3 for the layout of a 64-bit double-precision floating-point value; refer to Section 4.2.2., “Floating-Point Data Types” for a detailed description of double-precision floating-point values.)

• **128-bit packed integers.** The four 128-bit packed integer data types can contain 16 byte integers, 8 word integers, 4 doubleword integers, or 2 quadword integers. (Refer to Section 4.6.2., “128-Bit Packed SIMD Data Types” for a detailed description of the 128-bit packed integers.)

---

**Figure 11-2. Data Types Introduced with the SSE2 Extensions**

All of these data types are operated on in XMM registers or memory. Instructions are provided to convert between these 128-bit data types and the 64-bit and 32-bit data types. The address of a 128-bit packed memory operand must be aligned on a 16-byte boundary, except in the following cases:

• a MOVUPD instruction which supports unaligned accesses

• scalar instructions that use an 8-byte memory operand that is not subject to alignment requirements

Figure 4-2 shows the byte order of 128-bit (double quadword) and 64-bit (quadword) data types in memory.
11.4. SSE2 INSTRUCTIONS

The SSE2 instructions are divided into four functional groups:

- Packed and scalar double-precision floating-point instructions
- 64-bit and 128-bit SIMD integer instructions
- 128-bit extensions of SIMD integer instructions introduced with the MMX technology and the SSE extensions
- Cacheability-control and instruction-ordering instructions

The provide more information about each group.

11.4.1. Packed and Scalar Double-Precision Floating-Point Instructions

The packed and scalar double-precision floating-point instructions are divided into the following sub-groups:

- Data movement instructions
- Arithmetic instructions
- Comparison instructions
- Conversion instructions
- Logical instructions
- Shuffle instructions

The packed double-precision floating-point instructions perform SIMD operations similarly to the packed single-precision floating-point instructions (see Figure 11-3). Each source operand contains two double-precision floating-point values, and the destination operand contains the results of the operation (OP) performed in parallel on the corresponding values (X0 and Y0, and X1 and Y1) in each operand.
The scalar double-precision floating-point instructions operate on the low (least significant) quadwords of two source operands (X0 and Y0), as shown in Figure 11-4. The high quadword (X1) of the first source operand is passed through to the destination. The scalar operations are similar to the floating-point operations performed in x87 FPU data registers with the precision control field in the x87 FPU control word set for double precision (53-bit significand), except that x87 stack operations use a 15-bit exponent range for the result while SSE2 operations use an 11-bit exponent range.

See Section 11.6.8., “Compatibility of SIMD and x87 FPU Floating-Point Data Types” for more information about obtaining compatible results when performing both scalar double-precision floating-point operations in XMM registers and in x87 FPU data registers.
11.4.1.1. Data Movement Instructions

Data movement instructions move double-precision floating-point data between XMM registers and between XMM registers and memory.

The MOVAPD (move aligned packed double-precision floating-point) instruction transfers a 128-bit packed double-precision floating-point operand from memory to an XMM register or vice versa, or between XMM registers. The memory address must be aligned to a 16-byte boundary; if not, a general-protection exception (GP#) is generated.

The MOVUPD (move unaligned packed double-precision floating-point) instruction transfers a 128-bit packed double-precision floating-point operand from memory to an XMM register or vice versa, or between XMM registers. Alignment of the memory address is not required.

The MOVSD (move scalar double-precision floating-point) instruction transfers a 64-bit double-precision floating-point operand from memory to the low quadword of an XMM register or vice versa, or between XMM registers. Alignment of the memory address is not required, unless alignment checking is enabled.

The MOVHPD (move high packed double-precision floating-point) instruction transfers a 64-bit double-precision floating-point operand from memory to the high quadword of an XMM register or vice versa. The low quadword of the register is left unchanged. Alignment of the memory address is not required, unless alignment checking is enabled.

The MOVLPD (move low packed double-precision floating-point) instruction transfers a 64-bit double-precision floating-point operand from memory to the low quadword of an XMM register or vice versa. The high quadword of the register is left unchanged. Alignment of the memory address is not required, unless alignment checking is enabled.

The MOVMSKPD (move packed double-precision floating-point mask) instruction extracts the sign bit of each of the two packed double-precision floating-point numbers in an XMM register and saves them in a general-purpose register. This 2-bit value can then be used as a condition to perform branching.

11.4.1.2. SSE2 Arithmetic Instructions

SSE2 arithmetic instructions perform addition, subtraction, multiply, divide, square root, and maximum/minimum operations on packed and scalar double-precision floating-point values.

The ADDPD (add packed double-precision floating-point values) and SUBPD (subtract packed double-precision floating-point values) instructions add and subtract, respectively, two packed double-precision floating-point operands.

The ADDSD (add scalar double-precision floating-point values) and SUBSD (subtract scalar double-precision floating-point values) instructions add and subtract, respectively, the low double-precision floating-point values of two operands and stores the result in the low quadword of the destination operand.

The MULPD (multiply packed double-precision floating-point values) instruction multiplies two packed double-precision floating-point operands.
The MULSD (multiply scalar double-precision floating-point values) instruction multiplies the low double-precision floating-point values of two operands and stores the result in the low quadword of the destination operand.

The DIVPD (divide packed double-precision floating-point values) instruction divides two packed double-precision floating-point operands.

The DIVSD (divide scalar double-precision floating-point values) instruction divides the low double-precision floating-point values of two operands and stores the result in the low quadword of the destination operand.

The SQRTPD (compute square roots of packed double-precision floating-point values) instruction computes the square roots of the values in a packed double-precision floating-point operand.

The SQRTSD (compute square root of scalar double-precision floating-point values) instruction computes the square root of the low double-precision floating-point value in the source operand and stores the result in the low quadword of the destination operand.

The MAXPD (return maximum of packed double-precision floating-point values) instruction compares the corresponding values in two packed double-precision floating-point operands and returns the numerically greater value from each comparison to the destination operand.

The MAXSD (return maximum of scalar double-precision floating-point values) instruction compares the low double-precision floating-point values from two packed double-precision floating-point operands and returns the numerically higher value from the comparison to the low quadword of the destination operand.

The MINPD (return minimum of packed double-precision floating-point values) instruction compares the corresponding values from two packed double-precision floating-point operands and returns the numerically lesser value from each comparison to the destination operand.

The MINSD (return minimum of scalar double-precision floating-point values) instruction compares the low values from two packed double-precision floating-point operands and returns the numerically lesser value from the comparison to the low quadword of the destination operand.

### 11.4.1.3. SSE2 Logical Instructions

SSE2 logical instructions perform AND, AND NOT, OR, and XOR operations on packed double-precision floating-point values.

The ANDPD (bitwise logical AND of packed double-precision floating-point values) instruction returns the logical AND of two packed double-precision floating-point operands.

The ANDNPD (bitwise logical AND NOT of packed double-precision floating-point values) instruction returns the logical AND NOT of two packed double-precision floating-point operands.

The ORPD (bitwise logical OR of packed double-precision floating-point values) instruction returns the logical OR of two packed double-precision floating-point operands.
The XORPD (bitwise logical XOR of packed double-precision floating-point values) instruction returns the logical XOR of two packed double-precision floating-point operands.

**11.4.1.4. SSE2 Comparison Instructions**

SSE2 compare instructions compare packed and scalar double-precision floating-point values and return the results of the comparison either to the destination operand or to the EFLAGS register.

The CMPPD (compare packed double-precision floating-point values) instruction compares the corresponding values from two packed double-precision floating-point operands, using an immediate operand as a predicate, and returns a 64-bit mask result of all 1s or all 0s for each comparison to the destination operand. The value of the immediate operand allows the selection of any of eight compare conditions: equal, less than, less than equal, unordered, not equal, not less than, not less than or equal, or ordered.

The CMPSD (compare scalar double-precision floating-point values) instruction compares the low values from two packed double-precision floating-point operands, using an immediate operand as a predicate, and returns a 64-bit mask result of all 1s or all 0s for the comparison to the low quadword of the destination operand. The immediate operand selects the compare condition as with the CMPPD instruction.

The COMISD (compare scalar double-precision floating-point values and set EFLAGS) and UCOMISD (unordered compare scalar double-precision floating-point values and set EFLAGS) instructions compare the low values of two packed double-precision floating-point operands and set the ZF, PF, and CF flags in the EFLAGS register to show the result (greater than, less than, equal, or unordered). These two instructions differ as follows: the COMISD instruction signals a floating-point invalid-operation (#I) exception when a source operand is either a QNaN or an SNaN; the UCOMISD instruction only signals an invalid-operation exception when a source operand is an SNaN.

**11.4.1.5. SSE2 Shuffle and Unpack Instructions**

SSE2 shuffle instructions shuffle the contents of two packed double-precision floating-point values and store the results in the destination operand.

The SHUFPD (shuffle packed double-precision floating-point values) instruction places either of the two packed double-precision floating-point values from the destination operand in the low quadword of the destination operand, and places either of the two packed double-precision floating-point values from source operand in the high quadword of the destination operand (see Figure 11-5). By using the same register for the source and destination operands, the SHUFPD instruction can swap two packed double-precision floating-point values.
The UNPCKHPD (unpack and interleave high packed double-precision floating-point values) instruction performs an interleaved unpack of the high values from the source and destination operands and stores the result in the destination operand (see Figure 11-6).

The UNPCKLPD (unpack and interleave low packed double-precision floating-point values) instruction performs an interleaved unpack of the low values from the source and destination operands and stores the result in the destination operand (see Figure 11-7).
11.4.1.6. **SSE2 Conversion Instructions**

SSE2 conversion instructions (see Figure 11-8) support packed and scalar conversions between:
- Double-precision and single-precision floating-point formats
- Double-precision floating-point and doubleword integer formats
- Single-precision floating-point and doubleword integer formats

**Conversion between double-precision and single-precision floating-points values.** The following instructions convert operands between double-precision and single-precision floating-point formats. The operands being operated on are contained in XMM registers or memory (at most, one operand can reside in memory; the destination is always an MMX register).

The CVTPS2PD (convert packed single-precision floating-point values to packed double-precision floating-point values) instruction converts two packed single-precision floating-point values to two double-precision floating-point values.

The CVTPD2PS (convert packed double-precision floating-point values to packed single-precision floating-point values) instruction converts two packed double-precision floating-point values to two single-precision floating-point values. When a conversion is inexact, the result is rounded according to the rounding mode selected in the MXCSR register.

The CVTSS2SD (convert scalar single-precision floating-point value to scalar double-precision floating-point value) instruction converts a single-precision floating-point value to a double-precision floating-point value.

The CVTSD2SS (convert scalar double-precision floating-point value to scalar single-precision floating-point value) instruction converts a double-precision floating-point value to a single-precision floating-point value. When the conversion is inexact, the result is rounded according to the rounding mode selected in the MXCSR register.

---

**Figure 11-7. UNPCKLPD Instruction, Low Unpack and Interleave Operation**

<table>
<thead>
<tr>
<th>DEST</th>
<th>X1</th>
<th>X0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC</td>
<td>Y1</td>
<td>Y0</td>
</tr>
<tr>
<td>DEST</td>
<td>Y0</td>
<td>X0</td>
</tr>
</tbody>
</table>
Conversion between double-precision floating-point values and doubleword integers. The following instructions convert operands between double-precision floating-point and doubleword integer formats. Operands are housed in XMM registers, MMX registers, general registers or memory (at most one operand can reside in memory; the destination is always an XMM, MMX, or general register).

The CVTPD2PI (convert packed double-precision floating-point values to packed doubleword integers) instruction converts two packed double-precision floating-point numbers to two packed signed doubleword integers, with the result stored in an MMX register. When rounding to an integer value, the source value is rounded according to the rounding mode in the MXCSR register. The CVTTPD2PI (convert with truncation packed double-precision floating-point values to packed doubleword integers) instruction is similar to the CVTPD2PI instruction except that truncation is used to round a source value to an integer value (see Section 4.8.4.2., “Truncation with SSE and SSE2 Conversion Instructions”).

The CVTP12PD (convert packed doubleword integers to packed double-precision floating-point values) instruction converts two packed signed doubleword integers to two double-precision floating-point values.
The CVTPD2DQ (convert packed double-precision floating-point values to packed doubleword integers) instruction converts two packed double-precision floating-point numbers to two packed signed doubleword integers, with the result stored in the low quadword of an XMM register. When rounding an integer value, the source value is rounded according to the rounding mode selected in the MXCSR register. The CVTTPD2DQ (convert with truncation packed double-precision floating-point values to packed doubleword integers) instruction is similar to the CVTPD2DQ instruction except that truncation is used to round a source value to an integer value (see Section 4.8.4.2., “Truncation with SSE and SSE2 Conversion Instructions”).

The CVTDP2PD (convert packed doubleword integers to packed double-precision floating-point values) instruction converts two packed signed doubleword integers located in the low-order doublewords of an XMM register to two double-precision floating-point values.

The CVTSD2SI (convert scalar double-precision floating-point value to doubleword integer) instruction converts a double-precision floating-point value to a doubleword integer, and stores the result in a general-purpose register. When rounding an integer value, the source value is rounded according to the rounding mode selected in the MXCSR register. The CVTTS2SI (convert with truncation scalar double-precision floating-point value to doubleword integer) instruction is similar to the CVTSD2SI instruction except that truncation is used to round the source value to an integer value (see Section 4.8.4.2., “Truncation with SSE and SSE2 Conversion Instructions”).

The CVTSI2SD (convert doubleword integer to scalar double-precision floating-point value) instruction converts a signed doubleword integer in a general-purpose register to a double-precision floating-point number, and stores the result in an XMM register.

Conversion between single-precision floating-point and doubleword integer formats. These instructions convert between packed single-precision floating-point and packed doubleword integer formats. Operands are housed in XMM registers, MMX registers, general registers, or memory (the latter for at most one source operand). The destination is always an XMM, MMX, or general register. These SSE2 instructions supplement conversion instructions (CVTPS2SI, CVTSP2SI, CVTTPS2PI, CVTSI2SS, CVTSS2SI, and CVTTSS2SI) introduced with SSE extensions.

The CVTPS2DQ (convert packed single-precision floating-point values to packed doubleword integers) instruction converts four packed single-precision floating-point values to four packed signed doubleword integers, with the source and destination operands in XMM registers or memory (the latter for at most one source operand). When the conversion is inexact, the rounded value according to the rounding mode selected in the MXCSR register is returned. The CVTTPS2DQ (convert with truncation packed single-precision floating-point values to packed doubleword integers) instruction is similar to the CVTPS2DQ instruction except that truncation is used to round a source value to an integer value (see Section 4.8.4.2., “Truncation with SSE and SSE2 Conversion Instructions”).

The CVTDQ2PS (convert packed doubleword integers to packed single-precision floating-point values) instruction converts four packed signed doubleword integers to four packed single-precision floating-point numbers, with the source and destination operands in XMM registers or memory (the latter for at most one source operand). When the conversion is inexact, the rounded value according to the rounding mode selected in the MXCSR register is returned.
11.4.2. SSE2 64-Bit and 128-Bit SIMD Integer Instructions

SSE2 extensions add several 128-bit packed integer instructions to the IA-32 architecture. Where appropriate, a 64-bit version of each of these instructions is also provided. The 128-bit versions of instructions operate on data in XMM registers; 64-bit versions operate on data in MMX registers. The instructions follow.

The MOVDQA (move aligned double quadword) instruction transfers a double quadword operand from memory to an XMM register or vice versa; or between XMM registers. The memory address must be aligned to a 16-byte boundary; otherwise, a general-protection exception (#GP) is generated.

The MOVQDO (move unaligned double quadword) instruction performs the same operations as the MOVDAQ instruction, except that 16-byte alignment of a memory address is not required.

The PADDQ (packed quadword add) instruction adds two packed quadword integer operands or two single quadword integer operands, and stores the results in an XMM or MMX register, respectively. This instruction can operate on either unsigned or signed (two’s complement notation) integer operands.

The PSUBQ (packed quadword subtract) instruction subtracts two packed quadword integer operands or two single quadword integer operands, and stores the results in an XMM or MMX register, respectively. Like the PADDQ instruction, PSUBQ can operate on either unsigned or signed (two’s complement notation) integer operands.

The PMULUDQ (multiply packed unsigned doubleword integers) instruction performs an unsigned multiply of unsigned doubleword integers and returns a quadword result. Both 64-bit and 128-bit versions of this instruction are available. The 64-bit version operates on two doubleword integers stored in the low doubleword of each source operand, and the quadword result is returned to an MMX register. The 128-bit version performs a packed multiply of two pairs of doubleword integers. Here, the doublewords are packed in the first and third doublewords of the source operands, and the quadword results are stored in the low and high quadwords of an XMM register.

The PSHUFLW (shuffle packed low words) instruction shuffles the word integers packed into the low quadword of the source operand and stores the shuffled result in the low quadword of the destination operand. An 8-bit immediate operand specifies the shuffle order.

The PSHUFHW (shuffle packed high words) instruction shuffles the word integers packed into the high quadword of the source operand and stores the shuffled result in the high quadword of the destination operand. An 8-bit immediate operand specifies the shuffle order.

The PSHUFD (shuffle packed doubleword integers) instruction shuffles the doubleword integers packed into the source operand and stores the shuffled result in the destination operand. An 8-bit immediate operand specifies the shuffle order.

The PSLLDQ (shift double quadword left logical) instruction shifts the contents of the source operand to the left by the amount of bytes specified by an immediate operand. The empty low-order bytes are cleared (set to 0).
The PSRLDQ (shift double quadword right logical) instruction shifts the contents of the source operand to the right by the amount of bytes specified by an immediate operand. The empty high-order bytes are cleared (set to 0).

The PUNPCKHQDQ (Unpack high quadwords) instruction interleaves the high quadword of the source operand and the high quadword of the destination operand and writes them to the destination register.

The PUNPCKLQDQ (Unpack low quadwords) instruction interleaves the low quadwords of the source operand and the low quadwords of the destination operand and writes them to the destination register.

Two additional SSE instructions enable data movement from the MMX registers to the XMM registers.

The MOVQ2DQ (move quadword integer from MMX to XMM registers) instruction moves the quadword integer from an MMX source register to an XMM destination register.

The MOVDQ2Q (move quadword integer from XMM to MMX registers) instruction moves the low quadword integer from an XMM source register to an MMX destination register.

### 11.4.3. 128-Bit SIMD Integer Instruction Extensions

All of 64-bit SIMD integer instructions introduced with MMX technology and SSE extensions (with the exception of the PSHUFW instruction) have been extended by SSE2 extensions to operate on 128-bit packed integer operands located in XMM registers. The 128-bit versions of these instructions follow the same SIMD conventions regarding packed operands as the 64-bit versions. For example, where the 64-bit version of the PADDB instruction operates on 8 packed bytes, the 128-bit version operates on 16 packed bytes.

### 11.4.4. Cacheability Control and Memory Ordering Instructions

SSE2 extensions that give programs more control over the caching, loading, and storing of data are described below.

#### 11.4.4.1. FLUSH Cache Line

The CLFLUSH (flush cache line) instruction writes and invalidates the cache line associated with a specified linear address. The invalidation is for all levels of the processor’s cache hierarchy, and it is broadcast throughout the cache coherency domain.

Note that the CLFLUSH instruction was introduced with the SSE2 extensions, however, can be implemented in IA-32 processors that do not implement the SSE2 extensions. The CLFLUSH instruction has its own CPUID feature bit (bit 19 of register EDX) and can thus be detected independently from SSE2 extensions.
### 11.4.4.2. Cacheability Control Instructions

The following four instructions enable data from XMM and general-purpose registers to be stored to memory using a non-temporal hint. The non-temporal hint directs the processor to store data to memory without writing the data into the cache hierarchy whenever this is possible. See Section 10.4.6.2., “Caching of Temporal vs. Non-Temporal Data” for more information about non-temporal stores and hints.

- The MOVNTDQ (store double quadword using non-temporal hint) instruction stores packed integer data from an XMM register to memory, using a non-temporal hint.
- The MOVNTPD (store packed double-precision floating-point values using non-temporal hint) instruction stores packed double-precision floating-point data from an XMM register to memory, using a non-temporal hint.
- The MOVNTI (store doubleword using non-temporal hint) instruction stores integer data from a general-purpose register to memory, using a non-temporal hint.
- The MASKMOVQ (store selected bytes of double quadword) instruction stores selected byte integers from an XMM register to memory, using a byte mask to selectively write the individual bytes. The memory location does not need to be aligned on a natural boundary. This instruction also uses a non-temporal hint.

### 11.4.4.3. Memory Ordering Instructions

SSE2 extensions introduce two new fence instructions (LFENCE and MFENCE) as companions to the SFENCE instruction introduced with SSE extensions.

- The LFENCE instruction establishes a memory fence for loads. It guarantees ordering between two loads and prevents speculative loads from passing the load fence (that is, no speculative loads are allowed until all loads specified before the load fence have been carried out).
- The MFENCE instruction combines the functions of LFENCE and SFENCE by establishing a memory fence for both loads and stores. It guarantees that all loads and stores specified before the fence are globally observable prior to any loads or stores being carried out after the fence.

### 11.4.4.4. Pause

The PAUSE instruction is provided to improve the performance of “spin-wait loops” executed on a Pentium 4 or Intel Xeon processor. On a Pentium 4 processor, it also provides the added benefit of reducing processor power consumption while executing a spin-wait loop. It is recommended that a PAUSE instruction always be included in the code sequence for a spin-wait loop.
11.4.5. Branch Hints

SSE2 extensions designate two instruction prefixes (2EH and 3EH) to provide branch hints to the processor (see “Instruction Prefixes” in Chapter 2 of the IA-32 Intel Architecture Software Developer’s Manual, Volume 2A). These prefixes can only be used with the Jcc instruction and only at the machine code level (that is, there are no mnemonics for the branch hints).

11.5. SSE, SSE2, AND SSE3 EXCEPTIONS

SSE/SSE2/SSE3 extensions generate two general types of exceptions:

• Non-numeric exceptions
• SIMD floating-point exceptions

SSE/SSE2/SSE3 instructions can generate the same type of memory-access and non-numeric exceptions as other IA-32 architecture instructions. Existing exception handlers can generally handle these exceptions without any code modification. See “Providing Non-Numeric Exception Handlers for Exceptions Generated by the SSE, SSE2 and SSE3 Instructions” in Chapter 12, SSE, SSE2 and SSE3 System Programming, of the IA-32 Intel Architecture Software Developer’s Manual, Volume 3, for a list of the non-numeric exceptions that can be generated by SSE/SSE2/SSE3 instructions and for guidelines for handling these exceptions.

SSE/SSE2/SSE3 instructions do not generate numeric exceptions on packed integer operations; however, they can generate numeric (SIMD floating-point) exceptions on packed single-precision and double-precision floating-point operations. These SIMD floating-point exceptions are defined in the IEEE Standard 754 for Binary Floating-Point Arithmetic and are the same exceptions that are generated for x87 FPU instructions. See Section 11.5.1., “SIMD Floating-Point Exceptions” for a description of these exceptions.

11.5.1. SIMD Floating-Point Exceptions

SIMD floating-point exceptions are those exceptions that can be generated by SSE/SSE2/SSE3 instructions that operate on packed or scalar floating-point operands.

Six classes of SIMD floating-point exceptions can be generated:

• Invalid operation (#I)
• Divide-by-zero (#Z)
• Denormal operand (#D)
• Numeric overflow (#O)
• Numeric underflow (#U)
• Inexact result (Precision) (#P)

1. The FISTTP instruction in SSE3 does not generate SIMD floating-point exceptions, but it can generate x87 FPU floating-point exceptions.
All of these exceptions (except the denormal operand exception) are defined in IEEE Standard 754, and they are the same exceptions that are generated with the x87 floating-point instructions. Section 4.9., “Overview of Floating-Point Exceptions” gives a detailed description of these exceptions and of how and when they are generated. The following sections discuss the implementation of these exceptions in SSE/SSE2/SSE3 extensions.

All SIMD floating-point exceptions are precise and occur as soon as the instruction completes execution.

Each of the six exception conditions has a corresponding flag (IE, DE, ZE, OE, UE, and PE) and mask bit (IM, DM, ZM, OM, UM, and PM) in the MXCSR register (see Figure 10-3). The mask bits can be set with the LDMXCSR or FXRSTOR instruction; the mask and flag bits can be read with the STMXCSR or FXSAVE instruction.

The OSXMMEXCEPT flag (bit 10) of control register CR4 provides additional control over generation of SIMD floating-point exceptions by allowing the operating system to indicate whether or not it supports software exception handlers for SIMD floating-point exceptions. If an unmasked SIMD floating-point exception is generated and the OSXMMEXCEPT flag is set, the processor invokes a software exception handler by generating an SIMD floating-point exception (#XF). If the OSXMMEXCEPT bit is clear, the processor generates an invalid-opcode exception (#UD) on the first SSE or SSE2 instruction that detects an SIMD floating-point exception condition. See Section 11.6.2., “Checking for SSE and SSE2 Support”.

### 11.5.2. SIMD Floating-Point Exception Conditions

The following sections describe the conditions that cause an SIMD floating-point exception to be generated and the masked response of the processor when these conditions are detected.

See Section 4.9.2., “Floating-Point Exception Priority” for a description of the rules for exception precedence when more than one floating-point exception condition is detected for an instruction.

#### 11.5.2.1. Invalid Operation Exception (#I)

The floating-point invalid-operation exception (#I) occurs in response to an invalid arithmetic operand. The flag (IE) and mask (IM) bits for the invalid operation exception are bits 0 and 7, respectively, in the MXCSR register.

If the invalid-operation exception is masked, the processor returns a QNaN, QNaN floating-point indefinite, integer indefinite, one of the source operands to the destination operand, or it sets the EFLAGS, depending on the operation being performed. When a value is returned to the destination operand, it overwrites the destination register specified by the instruction. Table 11-1 lists the invalid-arithmetic operations that the processor detects for instructions and the masked responses to these operations.
If the invalid operation exception is not masked, a software exception handler is invoked and the operands remain unchanged. (See Section 11.5.4., “Handling SIMD Floating-Point Exceptions in Software”.)

Normally, when one or more of the source operands are QNaNs (and neither is an SNaN or in an unsupported format), an invalid-operation exception is not generated. The following instructions are exceptions to this rule: the COMISS and COMISD instructions; and the CMPSS, CMPPS, CMPPD, and CMPSD instructions (when the predicate is less than, less-than or equal, not less-than, or not less-than or equal). With these instructions, a QNaN source operand will generate an invalid-operation exception.

The invalid-operation exception is not affected by the flush-to-zero mode or by the denormals-are-zeros mode.

**Table 11-1. Masked Responses of SSE/SSE2/SSE3 Instructions to Invalid Arithmetic Operations**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Masked Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPS, ADDSS, ADDPD, ADDSD, SUBPS, SUBSS, SUBPD, SUBSD, MULPS, MULSS, MULPD, MULSD, DIVPS, DIVSS, DIVPD, DIVSD, ADDSUBPD, ADDSUBPS, HADDPS, HSUBPD or HSUBPS instruction with an SNaN operand</td>
<td>Return the SNaN converted to a QNaN; Refer to Table 4-7 for more details</td>
</tr>
<tr>
<td>SQRTPS, SQRTSS, SQRTPD, or SQRTPS with SNaN operands</td>
<td>Return the SNaN converted to a QNaN</td>
</tr>
<tr>
<td>SQRTPS, SQRTSS, SQRTPD, or SQRTPS with negative operands (except zero)</td>
<td>Return the QNaN floating-point Indefinite</td>
</tr>
<tr>
<td>CMPPS, CMPSS, CMPPD or CMPSD instruction with QNaN or SNaN operands</td>
<td>Return a mask of all 0s (except for the predicates “not-equal,” “unordered,” “not-less-than,” or “not-less-than-or-equal,” which returns a mask of all 1s)</td>
</tr>
<tr>
<td>CVTPS2PS, CVTSS2PS, CVTPS2PS, CVTSS2PS with SNaN operands</td>
<td>Return the SNaN converted to a QNaN</td>
</tr>
<tr>
<td>COMISS or COMISD with QNaN or SNaN operand(s)</td>
<td>Set EFLAGS values to “not comparable”</td>
</tr>
<tr>
<td>Addition of opposite signed infinities or subtraction of like-signed infinities</td>
<td>Return the QNaN floating-point Indefinite</td>
</tr>
<tr>
<td>Multiplication of infinity by zero</td>
<td>Return the QNaN floating-point Indefinite</td>
</tr>
<tr>
<td>Divide of (0/0) or (( \infty / \infty ) )</td>
<td>Return the QNaN floating-point Indefinite</td>
</tr>
<tr>
<td>Conversion to integer when the value in the source register is a NaN, ( \in), or exceeds the representable range for CVTPS2PI, CVTTPS2PI, CVTSS2SI, CVTTPPS2SI, CVTPS2DI, CVTTP2DI, CVTSS2DI, CVTTP2DI, CVTTPS2DI, CVTTP2DQ, CVTTPS2DQ, or CVTTPS2DI</td>
<td>Return the integer Indefinite</td>
</tr>
</tbody>
</table>
11.5.2.2. Denormal-Operand Exception (#D)

The processor signals the denormal-operand exception if an arithmetic instruction attempts to operate on a denormal operand. The flag (DE) and mask (DM) bits for the denormal-operand exception are bits 1 and 8, respectively, in the MXCSR register.

The CVTPD2I, CVTPD2PI, CVTTDQ2PD, CVTTPD2DQ, CVTSD2SD, CVTSD2SI, CVTS2SD, CVTSP2PS, CVTSP2PI, CVTS2SI, CVTSS2SI, CVTS2SS, CVTDQ2PS, CVTPS2DQ, and CVTTPS2DQ conversion instructions do not signal denormal exceptions. The RCPSS, RCPPS, RSQRTSS, and RSQRTPS instructions do not signal any kind of floating-point exception.

The denormals-are-zero flag (bit 6) of the MXCSR register provides an additional option for handling denormal-operand exceptions. When this flag is set, denormal source operands are automatically converted to zeros with the sign of the source operand (see Section 10.2.2.4., “Denormals-Are-Zeros”). The denormal operand exception is not affected by the flush-to-zero mode.

See Section 4.9.1.2., “Denormal Operand Exception (#D)” for more information about the denormal exception. See Section 11.5.4., “Handling SIMD Floating-Point Exceptions in Software” for information on handling unmasked exceptions.

11.5.2.3. Divide-By-Zero Exception (#Z)

The processor reports a divide-by-zero exception when a DIVPS, DIVSS, DIVPD or DIVSD instruction attempts to divide a finite non-zero operand by 0. The flag (ZE) and mask (ZM) bits for the divide-by-zero exception are bits 2 and 9, respectively, in the MXCSR register.

See Section 4.9.1.3., “Divide-By-Zero Exception (#Z)” for more information about the divide-by-zero exception. See Section 11.5.4., “Handling SIMD Floating-Point Exceptions in Software” for information on handling unmasked exceptions.

The divide-by-zero exception is not affected by the flush-to-zero mode or by the denormals-are-zeros mode.

11.5.2.4. Numeric Overflow Exception (#O)

The processor reports a numeric overflow exception whenever the rounded result of an arithmetic instruction exceeds the largest allowable finite value that fits in the destination operand. This exception can be generated with the ADDPS, ADDSS, ADDPD, ADDSD, SUBPS, SUBSS, SUBPD, SUBSD, MULPS, MULSS, MULPD, MULSD, DIVPS, DIVSS, DIVPD, DIVSD, CVTTPD2PS, CVTSD2SS, ADDSUBPD, ADDSUBPS, HADDPS, HADDPD, HADDP, HSUBPD and HSUBPS instructions. The flag (OE) and mask (OM) bits for the numeric overflow exception are bits 3 and 10, respectively, in the MXCSR register.

See Section 4.9.1.4., “Numeric Overflow Exception (#O)” for more information about the numeric-overflow exception. See Section 11.5.4., “Handling SIMD Floating-Point Exceptions in Software” for information on handling unmasked exceptions.
The numeric overflow exception is not affected by the flush-to-zero mode or by the denormals-are-zeros mode.

11.5.2.5. Numeric Underflow Exception (#U)

The processor reports a numeric underflow exception whenever the rounded result of an arithmetic instruction is less than the smallest possible normalized, finite value that will fit in the destination operand and the numeric-underflow exception is not masked. If the numeric underflow exception is masked, both underflow and the inexact-result condition must be detected before numeric underflow is reported. This exception can be generated with the ADDPS, ADDSS, ADDPD, ADDSD, SUBPS, SUBSS, SUBPD, SUBSD, MULPS, MULSS, MULPD, MULSD, DIVPS, DIVSS, DIVPD, DIVSD, CVTPD2PS, CVTSD2SS, ADDSUBPD, ADDSUBPS, HADDPD, HADDPS, HSUBPD, and HSUBPS instructions. The flag (UE) and mask (UM) bits for the numeric underflow exception are bits 4 and 11, respectively, in the MXCSR register.

The flush-to-zero flag (bit 15) of the MXCSR register provides an additional option for handling numeric underflow exceptions. When this flag is set and the numeric underflow exception is masked, tiny results (results that trigger the underflow exception) are returned as a zero with the sign of the true result (see Section 10.2.2.3., “Flush-To-Zero”). The numeric underflow exception is not affected by the denormals-are-zero mode.

See Section 4.9.1.5., “Numeric Underflow Exception (#U)” for more information about the numeric underflow exception. See Section 11.5.4., “Handling SIMD Floating-Point Exceptions in Software” for information on handling unmasked exceptions.

11.5.2.6. Inexact-Result (Precision) Exception (#P)

The inexact-result exception (also called the precision exception) occurs if the result of an operation is not exactly representable in the destination format. For example, the fraction 1/3 cannot be precisely represented in binary form. This exception occurs frequently and indicates that some (normally acceptable) accuracy has been lost. The exception is supported for applications that need to perform exact arithmetic only. Because the rounded result is generally satisfactory for most applications, this exception is commonly masked.

The flag (PE) and mask (PM) bits for the inexact-result exception are bits 2 and 12, respectively, in the MXCSR register.

See Section 4.9.1.6., “Inexact-Result (Precision) Exception (#P)” for more information about the inexact-result exception. See Section 11.5.4., “Handling SIMD Floating-Point Exceptions in Software” for information on handling unmasked exceptions.

In flush-to-zero mode, the inexact result exception is reported. The inexact result exception is not affected by the denormals-are-zero mode.
11.5.3. Generating SIMD Floating-Point Exceptions

When the processor executes a packed or scalar floating-point instruction, it looks for and reports on SIMD floating-point exception conditions using two sequential steps:

1. Looks for, reports on, and handles pre-computation exception conditions (invalid-operand, divide-by-zero, and denormal operand)
2. Looks for, reports on, and handles post-computation exception conditions (numeric overflow, numeric underflow, and inexact result)

If both pre- and post-computational exceptions are unmasked, it is possible for the processor to generate an SIMD floating-point exception (#XF) twice during the execution of an SSE, SSE2 or SSE3 instruction: once when it detects and handles a pre-computational exception and when it detects a post-computational exception.

11.5.3.1. Handling Masked Exceptions

If all exceptions are masked, the processor handles the exceptions it detects by placing the masked result (or results for packed operands) in a destination operand and continuing program execution. The masked result may be a rounded normalized value, signed infinity, a denormal finite number, zero, a QNaN floating-point indefinite, or a QNaN depending on the exception condition detected. In most cases, the corresponding exception flag bit in MXCSR is also set. The one situation where an exception flag is not set is when an underflow condition is detected and it is not accompanied by an inexact result.

When operating on packed floating-point operands, the processor returns a masked result for each of the sub-operand computations and sets a separate set of internal exception flags for each computation. It then performs a logical-OR on the internal exception flag settings and sets the exception flags in the MXCSR register according to the results of OR operations.

For example, Figure 11-9 shows the results of an MULPS instruction. In the example, all SIMD floating-point exceptions are masked. Assume that a denormal exception condition is detected prior to the multiplication of sub-operands X0 and Y0, no exception condition is detected for the multiplication of X1 and Y1, a numeric overflow exception condition is detected for the multiplication of X2 and Y2, and another denormal exception is detected prior to the multiplication of sub-operands X3 and Y3. Because denormal exceptions are masked, the processor uses the denormal source values in the multiplications of (X0 and Y0) and of (X3 and Y3) passing the results of the multiplications through to the destination operand. With the denormal operand, the result of the X0 and Y0 computation is a normalized finite value, with no exceptions detected. However, the X3 and Y3 computation produces a tiny and inexact result. This causes the corresponding internal numeric underflow and inexact-result exception flags to be set.
For the multiplication of X2 and Y2, the processor stores the floating-point $\infty$ in the destination operand, and sets the corresponding internal sub-operand numeric overflow flag. The result of the X1 and Y1 multiplication is passed through to the destination operand, with no internal sub-operand exception flags being set. Following the computations, the individual sub-operand exceptions flags for denormal operand, numeric underflow, inexact result, and numeric overflow are OR’d and the corresponding flags are set in the MXCSR register.

The net result of this computation is that:

- Multiplication of X0 and Y0 produces a normalized finite result
- Multiplication of X1 and Y1 produces a normalized finite result
- Multiplication of X2 and Y2 produces a floating-point $\infty$ result
- Multiplication of X3 and Y3 produces a tiny, inexact, finite result
- Denormal operand, numeric underflow, numeric underflow, and inexact result flags are set in the MXCSR register

11.5.3.2. Handling Unmasked Exceptions

If all exceptions are unmasked, the processor:

1. First detects any pre-computation exceptions: it ORs those exceptions, sets the appropriate exception flags, leaves the source and destination operands unaltered, and goes to step 2. If it does not detect any pre-computation exceptions, it goes to step 5.

2. Checks the OSXMMEXCPT flag (bit 10) of control register CR4. If this flag is set, the processor goes to step 3; if the flag is clear, it generates an invalid-opcode exception (#UD) and makes an implicit call to the invalid-opcode exception handler.

3. Generates an SIMD floating-point exception (#XF) and makes an implicit call to the SIMD floating-point exception handler.
4. If the exception handler is able to fix the source operands that generated the pre-computation exceptions or mask the condition in such a way as to allow the processor to continue executing the instruction, the processor resumes instruction execution as described in step 5.

5. Upon returning from the exception handler (or if no pre-computation exceptions were detected), the processor checks for post-computation exceptions. If the processor detects any post-computation exceptions: it ORs those exceptions, sets the appropriate exception flags, leaves the source and destination operands unaltered, and repeats steps 2, 3, and 4.

6. Upon returning from the exceptions handler in step 4 (or if no post-computation exceptions were detected), the processor completes the execution of the instruction.

The implication of this procedure is that for unmasked exceptions, the processor can generate an SIMD floating-point exception (#XF) twice: once if it detects pre-computation exception conditions and a second time if it detects post-computation exception conditions. For example, if SIMD floating-point exceptions are unmasked for the computation shown in Figure 11-9, the processor would generate one SIMD floating-point exception for denormal operand conditions and a second SIMD floating-point exception for overflow and underflow (no inexact result exception would be generated because the multiplications of X0 and Y0 and of X1 and Y1 are exact).

11.5.3.3. Handling Combinations of Masked and Unmasked Exceptions

In situations where both masked and unmasked exceptions are detected, the processor will set exception flags for the masked and the unmasked exceptions. However, it will not return masked results until after the processor has detected and handled unmasked post-computation exceptions and returned from the exception handler (as in step 6 above) to finish executing the instruction.

11.5.4. Handling SIMD Floating-Point Exceptions in Software

Section 4.9.3., “Typical Actions of a Floating-Point Exception Handler” shows actions that may be carried out by an SIMD floating-point exception handler. The SSE/SSE2/SSE3 state is saved with the FXSAVE instruction (see Section 11.6.5., “Saving and Restoring the SSE and SSE2 State”).

11.5.5. Interaction of SIMD and x87 FPU Floating-Point Exceptions

SIMD floating-point exceptions are generated independently from x87 FPU floating-point exceptions. SIMD floating-point exceptions do not cause assertion of the FERR# pin (independent of the value of CR0.NE). They ignore the assertion and deassertion of the IGNNE# pin.

If applications use SSE/SSE2/SSE3 instructions along with x87 FPU instructions (in the same task or program), consider the following:
• SIMD floating-point exceptions are reported independently from the x87 FPU floating-point exceptions. SIMD and x87 FPU floating-point exceptions can be unmasked independently. Separate x87 FPU and SIMD floating-point exception handlers must be provided if the same exception is unmasked for x87 FPU and for SSE/SSE2/SSE3 operations.

• The rounding mode specified in the MXCSR register does not affect x87 FPU instructions. Likewise, the rounding mode specified in the x87 FPU control word does not affect the SSE/SSE2/SSE3 instructions. To use the same rounding mode, the rounding control bits in the MXCSR register and in the x87 FPU control word must be set explicitly to the same value.

• The flush-to-zero mode set in the MXCSR register for SSE/SSE2/SSE3 instructions has no counterpart in the x87 FPU. For compatibility with the x87 FPU, set the flush-to-zero bit to 0.

• The denormals-are-zeros mode set in the MXCSR register for SSE/SSE2/SSE3 instructions has no counterpart in the x87 FPU. For compatibility with the x87 FPU, set the denormals-are-zeros bit to 0.

• An application that expects to detect x87 FPU exceptions that occur during the execution of x87 FPU instructions will not be notified if exceptions occur during the execution of corresponding SSE/SSE2/SSE3 instructions, unless the exception masks that are enabled in the x87 FPU control word have also been enabled in the MXCSR register and the application is capable of handling SIMD floating-point exceptions (#XF).
  — Masked exceptions that occur during an SSE/SSE2/SSE3 library call cannot be detected by unmasking the exceptions after the call (in an attempt to generate the fault based on the fact that an exception flag is set), an SIMD floating-point exception flag that is set when the corresponding exception is unmasked will not generate a fault; only the next occurrence of that unmasked exception will generate a fault.
  — An application which checks the x87 FPU status word to determine if any masked exception flags were set during an x87 FPU library call will also need to check the MXCSR register to detect a similar occurrence of a masked exception flag being set during an SSE/SSE2/SSE3 library call.

11.6. WRITING APPLICATIONS WITH THE SSE AND SSE2 EXTENSIONS

The following sections give some guidelines for writing application programs and operating-system code that uses the SSE and SSE2 extensions. Because SSE and SSE2 extensions share the same state and perform companion operations, these guidelines apply to both sets of extensions.

1. Here SSE3 refers to ADDSUBPD, ADDSUBPS, HADDPD, HADDPSS, HSUBPD and HSUBPS; the only other SSE3 instruction that can raise floating-point exceptions is FISTTP, which can generate x87 FPU invalid operation and inexact result exceptions.
Chapter 12, SSE, SSE2 and SSE3 System Programming in the IA-32 Intel Architecture Software Developer’s Manual, Volume 3 discusses the interface to the processor for context switching as well as other operating system considerations when writing code that uses SSE/SSE2/SSE3 extensions.

11.6.1. General Guidelines for Using the SSE and SSE2 Extensions

The following guidelines describe how to take full advantage of the performance gains available with the SSE and SSE2 extensions:

- Ensure that the processor supports the SSE and SSE2 extensions.
- Ensure that your operating system supports the SSE and SSE2 extensions. (Operating system support for the SSE extensions implies support for SSE2 extension and vice versa.)
- Use stack and data alignment techniques to keep data properly aligned for efficient memory use.
- Use the non-temporal store instructions offered with the SSE and SSE2 extensions.
- Employ the optimization and scheduling techniques described in the Intel Pentium 4 Optimization Reference Manual (see Section 1.4., “Related Literature”, for the order number for this manual).

11.6.2. Checking for SSE and SSE2 Support

Before an application attempts to use the SSE and/or SSE2 extensions, it should check that they are present on the processor and that the operating system supports them. The application can make this check by following these steps:

1. Check that the processor supports the CPUID instruction by attempting to execute the CPUID instruction. If the processor does not support the CPUID instruction, it will generate an invalid-opcode exception (#UD).
2. Check that the processor supports the SSE and/or SSE2 extensions. Execute the CPUID instruction with an argument of 1 in the EAX register, and check that bit 25 (SSE) and/or bit 26 (SSE2) are set to 1.
3. Check that the processor supports the FXSAVE and FXRSTOR instructions. Execute the CPUID instruction with an argument of 1 in the EAX register, and check that bit 24 (FXSR) is set to 1.
4. Check that the operating system supports the FXSAVE and FXRSTOR instructions. Execute a MOV instruction to read the contents of control register CR4, and check that bit 9 of CR4 (the OSFXSR bit) is set to 1.
5. Check that the operating system supports SIMD floating-point exception handling. Execute a MOV instruction to read the contents of control register CR4, and check that bit 10 of CR4 (the OSXMMEXCPT bit) is set to 1.
NOTE

The OSFXSR and OSXMMEXCPT bits in control register CR4 must be set by the operating system. The processor has no other way of detecting operating-system support for the FXSAVE and FXRSTOR instructions or for handling SIMD floating-point exceptions.

6. Check that emulation of the x87 FPU is disabled. Execute a MOV instruction to read the contents control register CR0, and check that bit 2 of CR0 (the EM bit) is set to 0.

If the processor attempts to execute an unsupported SSE or SSE2 instruction, the processor will generate an invalid-opcode exception (#UD).

11.6.3. Checking for the DAZ Flag in the MXCSR Register

The denormals-are-zero flag in the MXCSR register is available in most of the Pentium 4 processors and in the Intel Xeon processor, with the exception of some early steppings. To check for the presence of the DAZ flag in the MXCSR register, do the following:

1. Establish a 512-byte FXSAVE area in memory.
2. Clear the FXSAVE area to all 0s.
3. Execute the FXSAVE instruction, using the address of the first byte of the cleared FXSAVE area as a source operand. See “FXSAVE—Save x87 FPU, MMX, SSE, and SSE2 State” in Chapter 3 of the IA-32 Intel Architecture Software Developer’s Manual, Volume 2A for a description of the FXSAVE instruction and the layout of the FXSAVE image.
4. Check the value in the MXCSR_MASK field in the FXSAVE image (bytes 28 through 31).
   — If the value of the MXCSR_MASK field is 00000000H, the DAZ flag and denormals-are-zero mode are not supported.
   — If the value of the MXCSR_MASK field is non-zero and bit 6 is set, the DAZ flag and denormals-are-zero mode are supported.

If the DAZ flag is not supported, then it is a reserved bit and attempting to write a 1 to it will cause a general-protection exception (#GP). See Section 11.6.6., “Guidelines for Writing to the MXCSR Register” for general guidelines for preventing general-protection exceptions when writing to the MXCSR register.

11.6.4. Initialization of the SSE and SSE2 Extensions

The SSE and SSE2 state is contained in the XMM and MXCSR registers. Upon a hardware reset of the processor, this state is initialized as follows (see Table 11-2):

• All SIMD floating-point exceptions are masked (bits 7 through 12 of the MXCSR register is set to 1).
• All SIMD floating-point exception flags are cleared (bits 0 through 5 of the MXCSR register is set to 0).
• The rounding control is set to round-nearest (bits 13 and 14 of the MXCSR register are set to 00B).
• The flush-to-zero mode is disabled (bit 15 of the MXCSR register is set to 0).
• The denormals-are-zeros mode is disabled (bit 6 of the MXCSR register is set to 0). If the denormals-are-zeros mode is not supported, this bit is reserved and will be set to 0 on initialization.
• Each of the XMM registers is cleared (set to all zeros).

### Table 11-2. SSE and SSE2 State Following a Power-up/Reset or INIT

<table>
<thead>
<tr>
<th>Registers</th>
<th>Power-Up or Reset</th>
<th>INIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMM0 through XMM7</td>
<td>+0.0</td>
<td>Unchanged</td>
</tr>
<tr>
<td>MXCSR</td>
<td>1F80H</td>
<td>Unchanged</td>
</tr>
</tbody>
</table>

If the processor is reset by asserting the INIT# pin, the SSE and SSE2 state is not changed.

### 11.6.5. Saving and Restoring the SSE and SSE2 State

The FXSAVE instruction saves the x87 FPU, MMX, SSE and SSE2 states (which includes the contents of eight XMM registers and the MXCSR registers) in a 512-byte block of memory. The FXRSTOR instruction restores the saved SSE and SSE2 state from memory. See the FXSAVE instruction in Chapter 3, *Instruction Set Reference A-M*, of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 2A* for the layout of the 512-byte state block.

In addition to saving and restoring the SSE and SSE2 state, FXSAVE and FXRSTOR also save and restore the x87 FPU state (because MMX registers are aliased to the x87 FPU data registers this includes saving and restoring the MMX state). For greater code efficiency, it is suggested that FXSAVE and FXRSTOR be substituted for the FSAVE, FNSAVE and FRSTOR instructions in the following situations:

• When a context switch is being made in a multitasking environment
• During calls and returns from interrupt and exception handlers

In situations where the code is switching between x87 FPU and MMX technology computations (without a context switch or a call to an interrupt or exception), the FSAVE/FNSAVE and FRSTOR instructions are more efficient than the FXSAVE and FXRSTOR instructions.
11.6.6. Guidelines for Writing to the MXCSR Register

The MXCSR has several reserved bits, and attempting to write a 1 to any of these bits will cause a general-protection exception (#GP) to be generated. To allow software to identify these reserved bits, the MXCSR_MASK value is provided. Software can determine this mask value as follows:

1. Establish a 512-byte FXSAVE area in memory.
2. Clear the FXSAVE area to all 0s.
3. Execute the FXSAVE instruction, using the address of the first byte of the cleared FXSAVE area as a source operand. See “FXSAVE—Save x87 FPU, MMX, SSE, and SSE2 State” in Chapter 3 of the IA-32 Intel Architecture Software Developer’s Manual, Volume 2A for a description of FXSAVE and the layout of the FXSAVE image.
4. Check the value in the MXCSR_MASK field in the FXSAVE image (bytes 28 through 31).
   — If the value of the MXCSR_MASK field is 00000000H, then the MXCSR_MASK value is the default value of 0000FFBFH. Note that this value indicates that bit 6 of the MXCSR register is reserved; this setting indicates that the denormals-are-zero mode is not supported on the processor.
   — If the value of the MXCSR_MASK field is non-zero, the MXCSR_MASK value should be used as the MXCSR_MASK.

All bits set to 0 in the MXCSR_MASK value indicate reserved bits in the MXCSR register. Thus, if the MXCSR_MASK value is AND’d with a value to be written into the MXCSR register, the resulting value will be assured of having all its reserved bits set to 0, preventing the possibility of a general-protection exception being generated when the value is written to the MXCSR register.

For example, the default MXCSR_MASK value when 00000000H is returned in the FXSAVE image is 0000FFBFH. If software AND’s a value to be written to MXCSR register with 0000FFBFH, bit 6 of the result (the DAZ flag) will be ensured of being set to 0, which is the required setting to prevent general-protection exceptions on processors that do not support the denormals-are-zero mode.

To prevent general-protection exceptions, the MXCSR_MASK value should be AND’d with the value to be written into the MXCSR register in the following situations:

• Operating system routines that receive a parameter from an application program and then write that value to the MXCSR register (either with an FXRSTOR or LDMXCSR instruction)
• Any application program that writes to the MXCSR register and that needs to run robustly on several different IA-32 processors

Note that all bits in the MXCSR_MASK value that are set to 1 indicate features that are supported by the MXCSR register; so the bits in the MXCSR_MASK value can also treated as feature flags for identifying processor capabilities (similar to the feature flags information returned by the CPUID instruction).
11.6.7. Interaction of SSE and SSE2 Instructions with x87 FPU and MMX Instructions

The XMM registers and the x87 FPU and MMX registers represent separate execution environments, which has certain ramifications when executing SSE, SSE2, MMX, and x87 FPU instructions in the same code module or when mixing code modules that contain these instructions:

- Those SSE and SSE2 instructions that operate only on XMM registers (such as the packed and scalar floating-point instructions and the 128-bit SIMD integer instructions) in the same instruction stream with 64-bit SIMD integer or x87 FPU instructions without any restrictions. For example, an application can perform the majority of its floating-point computations in the XMM registers, using the packed and scalar floating-point instructions, and at the same time use the x87 FPU to perform trigonometric and other transcendental computations. Likewise, an application can perform packed 64-bit and 128-bit SIMD integer operations together without restrictions.

- Those SSE and SSE2 instructions that operate on MMX registers (such as the CVTPS2PI, CVTTPS2PI, CVTPS2PS, CVTPD2PS, CVTTPD2PI, CVTPD2PD, MOVQ2DQ, MOVQ2Q, PADADDQ, and PSUBQ instructions) can also be executed in the same instruction stream as 64-bit SIMD integer or x87 FPU instructions, however, here they are subject to the restrictions on the simultaneous use of MMX technology and x87 FPU instructions, which include:
  — Transition from x87 FPU to MMX technology instructions or to SSE or SSE2 instructions that operate on MMX registers should be preceded by saving the state of the x87 FPU.
  — Transition from MMX technology instructions or from SSE or SSE2 instructions that operate on MMX registers to x87 FPU instructions should be preceded by execution of the EMMS instruction.

11.6.8. Compatibility of SIMD and x87 FPU Floating-Point Data Types

SSE and SSE2 extensions operate on the same single-precision and double-precision floating-point data types that the x87 FPU operates on. However, when operating on these data types, the SSE and SSE2 extensions operate on them in their native format (single-precision or double-precision), in contrast to the x87 FPU which extends them to double extended-precision floating-point format to perform computations and then rounds the result back to a single-precision or double-precision format before writing results to memory. Because the x87 FPU operates on a higher precision format and then rounds the result to a lower precision format, it may return a slightly different result when performing the same operation on the same single-precision or double-precision floating-point values than is returned by the SSE and SSE2 extensions. The difference occurs only in the least-significant bits of the significand.
11.6.9. Intermixing Packed and Scalar Floating-Point and 128-Bit SIMD Integer Instructions and Data

SSE and SSE2 extensions define typed operations on packed and scalar floating-point data types and on 128-bit SIMD integer data types, but IA-32 processors do not enforce this typing at the architectural level. They only enforce it at the microarchitectural level. Therefore, when a Pentium 4 or Intel Xeon processor loads a packed or scalar floating-point operand or a 128-bit packed integer operand from memory into an XMM register, it does not check that the actual data being loaded matches the data type specified in the instruction. Likewise, when the processor performs an arithmetic operation on the data in an XMM register, it does not check that the data being operated on matches the data type specified in the instruction.

As a general rule, because data typing of SIMD floating-point and integer data types is not enforced at the architectural level, it is the responsibility of the programmer, assembler, or compiler to insure that code enforces data typing. Failure to enforce correct data typing can lead to computations that return unexpected results.

For example, in the following code sample, two packed single-precision floating-point operands are moved from memory into XMM registers (using MOVAPS instructions); then a double-precision packed add operation (using the ADDPD instruction) is performed on the operands:

```
movaps xmm0, [eax] ; EAX register contains pointer to packed
                   ; single-precision floating-point operand
movaps xmm1, [ebx]
addpd xmm0, xmm1
```

Pentium 4 and Intel Xeon processors execute these instructions without generating an invalid-operand exception (#UD) and will produce the expected results in register XMM0 (that is, the high and low 64-bits of each register will be treated as a double-precision floating-point value and the processor will operate on them accordingly). Because the data types operated on and the data type expected by the ADDPD instruction were inconsistent, the instruction may result in an SIMD floating-point exception (such as numeric overflow [#O] or invalid operation [#I]) being generated, but the actual source of the problem (inconsistent data types) is not detected.

The ability to operate on an operand that contains a data type that is inconsistent with the typing of the instruction being executed, permits some valid operations to be performed. For example, the following instructions load a packed double-precision floating-point operand from memory to register XMM0, and a mask to register XMM1; then they use XORPD to toggle the sign bits of the two packed values in register XMM0.

```
movapd xmm0, [eax] ; EAX register contains pointer to packed
                   ; double-precision floating-point operand
movaps xmm1, [ebx] ; EBX register contains pointer to packed
                   ; double-precision floating-point mask
xorpd xmm0, xmm1 ; XOR operation toggles sign bits using
                 ; the mask in xmm1
```
In this example: XORPS or PXOR can be used in place of XORPD and yield the same correct result. However, because of the type mismatch between the operand data type and the instruction data type, a latency penalty will be incurred due to implementations of the instructions at the microarchitecture level.

Latency penalties can also be incurred by using move instructions of the wrong type. For example, MOVAPS and MOVAPD can both be used to move a packed single-precision operand from memory to an XMM register. However, if MOVAPD is used, a latency penalty will be incurred when a correctly typed instruction attempts to use the data in the register.

Note that these latency penalties are not incurred when moving data from XMM registers to memory.

11.6.10. Interfacing with SSE and SSE2 Procedures and Functions

SSE and SSE2 extensions allow direct access to XMM registers. This means that all existing interface conventions between procedures and functions that apply to the use of the general-purpose registers (EAX, EBX, etc.) also apply to XMM register usage.

11.6.10.1. Passing Parameters in XMM Registers

The state of XMM registers is preserved across procedure (or function) boundaries. Parameters can be passed from one procedure to another using XMM registers.

11.6.10.2. Saving XMM Register State on a Procedure or Function Call

The state of XMM registers can be saved in two ways: using an FXSAVE instruction or a move instruction. FXSAVE saves the state of all XMM registers (along with the state of MXCSR and the x87 FPU registers). This instruction is typically used for major changes in the context of the execution environment, such as a task switch. FXRSTOR restores the XMM, MXCSR, and x87 FPU registers stored with FXSAVE.

In cases where only XMM registers must be saved, or where selected XMM registers need to be saved, move instructions (MOVAPS, MOVUPS, MOVSS, MOVAPD, MOVUPD, MOVSD, MOVQ, and MOVQ) can be used. These instructions can also be used to restore the contents of XMM registers. To avoid performance degradation when saving XMM registers to memory or when loading XMM registers from memory, be sure to use the appropriately typed move instructions.

The move instructions can also be used to save the contents of XMM registers on the stack. Here, the stack pointer (in the ESP register) can be used as the memory address to the next available byte in the stack. Note that the stack pointer is not automatically incremented when using a move instruction (as it is with PUSH).
A move-instruction procedure that saves the contents of an XMM register to the stack is responsible for decrementing the value in the ESP register by 16. Likewise, a move-instruction procedure that loads an XMM register from the stack needs also to increment the ESP register by 16. To avoid performance degradation when moving the contents of XMM registers, use the appropriately typed move instructions.

Use the LDMXCSR and STMXCSR instructions to save and restore, respectively, the contents of the MXCSR register on a procedure call and return.

11.6.10.3. Caller-Save Requirement for Procedure and Function Calls

When making procedure (or function) calls from SSE or SSE2 code, a caller-save convention is recommended for saving the state of the calling procedure. Using this convention, any register whose content must survive intact across a procedure call must be stored in memory by the calling procedure prior to executing the call.

The primary reason for using the caller-save convention is to prevent performance degradation. XMM registers can contain packed or scalar double-precision floating-point, packed single-precision floating-point, and 128-bit packed integer data types. The called procedure has no way of knowing the data types in XMM registers following a call; so it is unlikely to use the correctly typed move instruction to store the contents of XMM registers in memory or to restore the contents of XMM registers from memory.

As described in Section 11.6.9., “Intermixing Packed and Scalar Floating-Point and 128-Bit SIMD Integer Instructions and Data”, executing a move instruction that does not match the type for the data being moved to/from XMM registers will be carried out correctly, but can lead to a greater instruction latency.

11.6.11. Updating Existing MMX Technology Routines Using 128-Bit SIMD Integer Instructions

SSE2 extensions extend all 64-bit MMX SIMD integer instructions to operate on 128-bit SIMD integers using XMM registers. The extended 128-bit SIMD integer instructions operate like the 64-bit SIMD integer instructions; this simplifies the porting of MMX technology applications. However, there are considerations:

- To take advantage of wider 128-bit SIMD integer instructions, MMX technology code must be recompiled to reference the XMM registers instead of MMX registers.
- Computation instructions that reference memory operands that are not aligned on 16-byte boundaries should be replaced with an unaligned 128-bit load (MOVUDQ instruction) followed by a version of the same computation operation that uses register instead of memory operands. Use of 128-bit packed integer computation instructions with memory operands that are not 16-byte aligned results in a general protection exception (#GP).
- Extension of the PSHUFW instruction (shuffle word across 64-bit integer operand) across a full 128-bit operand is emulated by a combination of the following instructions: PSHUFHW, PSHUFLW, and PSHUFD.
Use of the 64-bit shift by bit instructions (PSRLQ, PSLLQ) can be extended to 128 bits in either of two ways:

- Use of PSRLQ and PSLLQ, along with masking logic operations.
- Rewriting the code sequence to use PSRLDQ and PSLLDQ (shift double quadword operand by bytes)

Loop counters need to be updated, since each 128-bit SIMD integer instruction operates on twice the amount of data as its 64-bit SIMD integer counterpart.


There are no condition codes in SSE or SSE2 states. A packed-data comparison instruction generates a mask which can then be transferred to an integer register. The following code sequence provides an example of how to perform a conditional branch, based on the result of an SSE2 arithmetic operation.

```c
cmppd XMM0, XMM1; generates a mask in XMM0
movmskpd EAX, XMM0; moves a 2 bit mask to eax
test EAX, 0,2 ; compare with desired result
jne BRANCH TARGET
```

The COMISD and UCOMISD instructions update the EFLAGS as the result of a scalar comparison. A conditional branch can then be scheduled immediately following COMISD/UCOMISD.

11.6.13. Cacheability Hint Instructions

SSE and SSE2 cacheability control instructions enable the programmer to control prefetching, caching, loading and storing of data. When correctly used, these instructions improve application performance.

To make efficient use of the processor’s super-scalar microarchitecture, a program needs to provide a steady stream of data to the executing program to avoid stalling the processor. PREFETCHh instructions minimize the latency of data accesses in performance-critical sections of application code by allowing data to be fetched into the processor cache hierarchy in advance of actual usage.

PREFETCHh instructions do not change the user-visible semantics of a program, although they may affect performance. The operation of these instructions is implementation-dependent. Programmers may need to tune code for each IA-32 processor implementation. Excessive usage of PREFETCHh instructions may waste memory bandwidth and reduce performance. For more detailed information on the use of prefetch hints, refer to Chapter 6, “Optimizing Cache Utilization for Pentium III Processors”, in the Pentium 4 Processor Optimization Reference Manual (see Section 1.4., “Related Literature” for the order number for this manual).

The non-temporal store instructions (MOVNTI, MOVNTPD, MOVNTPS, MOVNTDQ, MOVNTQ, MASKMOVQ, and MASKMOVDQU) minimize cache pollution when writing
non-temporal data to memory (see Section 10.4.6.2., “Caching of Temporal vs. Non-Temporal Data”, and Section 10.4.6.1., “Cacheability Control Instructions”). They prevent non-temporal data from being written into processor caches on a store operation. These instructions are implementation specific. Programmers may have to tune their applications for each IA-32 processor implementation to take advantage of these instructions.

Besides reducing cache pollution, the use of weakly-ordered memory types can be important under certain data sharing relationships, such as a producer-consumer relationship. The use of weakly ordered memory can make the assembling of data more efficient; but care must be taken to ensure that the consumer obtains the data that the producer intended. Some common usage models that may be affected in this way by weakly-ordered stores are:

- Library functions that use weakly ordered memory to write results
- Compiler-generated code that writes weakly-ordered results
- Hand-crafted code

The degree to which a consumer of data knows that the data is weakly ordered can vary for these cases. As a result, the SFENCE or MFENCE instruction should be used to ensure ordering between routines that produce weakly-ordered data and routines that consume the data. SFENCE and MFENCE provide a performance-efficient way to ensure ordering by guaranteeing that every store instruction that precedes SFENCE/MFENCE in program order is globally visible before a store instruction that follows the fence.

### 11.6.14. Effect of Instruction Prefixes on the SSE and SSE2 Instructions

Table 11-3 describes the effects of instruction prefixes on SSE and SSE2 instructions. (Table 11-3 also applies to SIMD integer and SIMD floating-point instructions in SSE3.) Unpredictable behavior can range from prefixes being treated as a reserved operation on one generation of IA-32 processors to generating an invalid opcode exception on another generation of processors.

See also “Instruction Prefixes” in Chapter 2 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 2A*, for complete description of instruction prefixes.

**NOTE**

Some of the SSE/SSE2/SSE3 instructions have two-byte opcodes that are either 2 byte or 3 byte in length. The two-byte opcodes that are three bytes in length consists of 3 bytes in the following order: the first byte must be one of the three mandatory prefix F2H, F3H, or 66H; the second byte must be 0FH; the third byte is an opcode byte. As shown in Table 11-3, using operand size and repeat prefixes with SSE, SSE2 and SSE3 instructions is reserved.
Table 11-3. Effect of Prefixes on SSE, SSE2 and SSE3 Instructions

<table>
<thead>
<tr>
<th>Prefix Type</th>
<th>Effect on SSE, SSE2 and SSE3 Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Size Prefix (67H)</td>
<td>Affects instructions with a memory operand. Reserved for instructions without a memory operand and may result in unpredictable behavior.</td>
</tr>
<tr>
<td>Operand Size (66H)</td>
<td>Reserved and may result in unpredictable behavior.</td>
</tr>
<tr>
<td>Segment Override (2EH,36H,3EH,26H,64H,65H)</td>
<td>Affects instructions with a memory operand. Reserved for instructions without a memory operand and may result in unpredictable behavior.</td>
</tr>
<tr>
<td>Repeat Prefixes (F2H and F3H)</td>
<td>Reserved and may result in unpredictable behavior.</td>
</tr>
<tr>
<td>Lock Prefix (F0H)</td>
<td>Reserved; generates invalid opcode exception (#UD).</td>
</tr>
<tr>
<td>Branch Hint Prefixes (E2H and E3H)</td>
<td>Reserved and may result in unpredictable behavior.</td>
</tr>
</tbody>
</table>
Programming With the Streaming SIMD Extensions 3 (SSE3)
CHAPTER 12
PROGRAMMING WITH
STREAMING SIMD EXTENSIONS 3 (SSE3)

The Pentium® 4 processor supporting Hyper-Threading Technology introduces Streaming SIMD Extensions 3 (SSE3). This chapter describes SSE3 and provides information to assist in writing application programs that use these extensions.

12.1. OVERVIEW OF SSE3 INSTRUCTIONS

SSE3 extensions include 13 instructions. Ten of these 13 instructions support the single instruction multiple data (SIMD) execution model used with SSE/SSE2 extensions. One SSE3 instruction accelerates x87 style programming for conversion of a floating-point value to integer. The remaining two instructions (MONITOR and MWAIT) accelerate synchronization of threads.

For additional information, see:
• Section 12.3., SSE3 Instructions provides an introduction to individual SSE3 instructions.
• Chapter 12, SSE, SSE2 and SSE3 System Programming, in the IA-32 Intel Architecture Software Developer’s Manual, Volume 3 gives guidelines for integrating SSE/SSE2/SSE3 extensions into an operating-system environment.

12.2. SSE3 PROGRAMMING ENVIRONMENT AND DATA TYPES

The programming environment for using SSE3 extensions is unchanged from that shown in Figure 3-1 and Figure 11-1. SSE3 does not introduce new data types. XMM registers are used to operate on packed integer data, on single-precision floating-point data or on double-precision floating-point data.

The x87 FPU is used for x87-style programming. The general registers are used by SSE3 instructions for thread synchronization. The MXCSR register governs SIMD floating-point operations. Note, however, that the x87FPU control word does not affect the SSE3 instruction that is executed by the x87 FPU (FISTTP), other than by unmasking an invalid operand or inexact result exception.
12.2.1. Compatibility of SSE3 Extensions with MMX Technology, the x87 FPU Environment, SSE Extensions and SSE2 Extensions

SSE3 extensions do not introduce any new state to the IA-32 execution environment. For SIMD and x87 programming, the FXSAVE and FXRSTOR instructions save and restore the architectural states of XMM, MXCSR, x87 FPU, and MMX registers. The MONITOR and MWAIT instructions use general purpose registers on input, they do not modify the content of those registers.

12.2.2. Horizontal and Asymmetric Processing

Most SSE/SSE2 instructions accelerate SIMD data processing using a model referred to as vertical computation. Using this model, data flow is vertical between the data elements of the inputs and the output (Figure 10-5 provides an example). SSE3 introduces instructions that accelerate SIMD floating-point processing where the result of each data element of the output includes either asymmetric processing or horizontal data movement of the input data elements. Figure 12-1 illustrates the asymmetric processing of the SSE3 instruction ADDSUBPD. Figure 12-2 illustrates the horizontal data movement of the SSE3 instruction HADDPD.

![Asymmetric Processing in ADDSUBPD](image)

Figure 12-1. Asymmetric Processing in ADDSUBPD
12.3. **SSE3 INSTRUCTIONS**

SSE3 offers 13 instructions that accelerate performance of Streaming SIMD Extensions technology, Streaming SIMD Extensions 2 technology, and x87-FP math capabilities. These instructions are grouped as follows:

- **x87 FPU instruction**
  - One instruction that improves x87 FPU floating-point to integer conversion
- **SIMD integer instruction**
  - One instruction that provides a specialized 128-bit unaligned data load
- **SIMD floating-point instructions**
  - Three instructions that enhance LOAD/MOVE/DUPLICATE performance
  - Two instructions that provide packed addition/subtraction
  - Four instructions that provide horizontal addition/subtraction
- **Thread synchronization instructions**
  - Two instructions that improve synchronization between multi-threaded agents

The instructions are discussed in more detail in the following paragraphs.
12.3.1. **x87 FPU Instruction for Integer Conversion**

The FISTTP instruction (x87 FPU Store Integer and Pop with Truncation) behaves like FISTP, but uses truncation regardless of what rounding mode is specified in the x87 FPU control word. The instruction converts the top of stack (ST0) to integer with rounding to and pops the stack.

The FISTTP instruction is available in three precisions: short integer (word or 16-bit), integer (double word or 32-bit), and long integer (64-bit). With FISTTP, applications no longer need to change the FCW when truncation is required.

12.3.2. **SIMD Integer Instruction for Specialized 128-bit Unaligned Data Load**

The LDDQU instruction is a special 128-bit unaligned load designed to avoid cache line splits. If the address of a 16-byte load is on a 16-byte boundary, LDDQU loads the bytes requested. If the address of the load is not aligned on a 16-byte boundary, LDDQU loads a 32-byte block starting at the 16-byte aligned address immediately below the load request. It then extracts the requested 16 bytes.

The instruction provides significant performance improvement on 128-bit unaligned memory accesses at the cost of some usage model restrictions.

12.3.3. **SIMD Floating-Point Instructions That Enhance LOAD/MOVE/DUPLICATE Performance**

The MOVSHDUP instruction loads/moves 128-bits, duplicating the second and fourth 32-bit data elements.

- **MOVSHDUP OperandA, OperandB**
  - OperandA (128 bits, four data elements): 3a, 2a, 1a, 0a
  - OperandB (128 bits, four data elements): 3b, 2b, 1b, 0b
  - Result (stored in OperandA): 3b, 3b, 1b, 1b

The MOVSLDUP instruction loads/moves 128-bits, duplicating the first and third 32-bit data elements.

- **MOVSLDUP OperandA, OperandB**
  - OperandA (128 bits, four data elements): 3a, 2a, 1a, 0a
  - OperandB (128 bits, four data elements): 3b, 2b, 1b, 0b
  - Result (stored in OperandA): 2b, 2b, 0b, 0b
The MOVDDUP instruction loads/moves 64-bits; duplicating the 64 bits from the source.

- MOVDDUP OperandA, OperandB
  - OperandA (128 bits, two data elements): 1\textsubscript{a}, 0\textsubscript{a}
  - OperandB (64 bits, one data element): 0\textsubscript{b}
  - Result (stored in OperandA): 0\textsubscript{b}, 0\textsubscript{b}

12.3.4. SIMD Floating-Point Instructions Provide Packed Addition/Subtraction

The ADDSUBPS instruction has two 128-bit operands. The instruction performs single-precision addition on the second and fourth pairs of 32-bit data elements within the operands; and single-precision subtraction on the first and third pairs.

- ADDSUBPS OperandA, OperandB
  - OperandA (128 bits, four data elements): 3\textsubscript{a}, 2\textsubscript{a}, 1\textsubscript{a}, 0\textsubscript{a}
  - OperandB (128 bits, four data elements): 3\textsubscript{b}, 2\textsubscript{b}, 1\textsubscript{b}, 0\textsubscript{b}
  - Result (stored in OperandA): 3\textsubscript{a}+3\textsubscript{b}, 2\textsubscript{a}-2\textsubscript{b}, 1\textsubscript{a}+1\textsubscript{b}, 0\textsubscript{a}-0\textsubscript{b}

The ADDSUBPD instruction has two 128-bit operands. The instruction performs double-precision addition on the second pair of quadwords, and double-precision subtraction on the first pair.

- ADDSUBPD OperandA, OperandB
  - OperandA (128 bits, two data elements): 1\textsubscript{a}, 0\textsubscript{a}
  - OperandB (128 bits, two data elements): 1\textsubscript{b}, 0\textsubscript{b}
  - Result (stored in OperandA): 1\textsubscript{a}+1\textsubscript{b}, 0\textsubscript{a}-0\textsubscript{b}

12.3.5. SIMD Floating-Point Instructions Provide Horizontal Addition/Subtraction

Most SIMD instructions operate vertically. This means that the result in position i is a function of the elements in position i of both operands. Horizontal addition/subtraction operates horizontally. This means that contiguous data elements in the same source operand are used to produce a result.

The HADDPS instruction performs a single-precision addition on contiguous data elements. The first data element of the result is obtained by adding the first and second elements of the first operand; the second element by adding the third and fourth elements of the first operand; the third by adding the first and second elements of the second operand; and the fourth by adding the third and fourth elements of the second operand.
12.3.6. Two Thread Synchronization Instructions

The MONITOR instruction sets up an address range that is used to monitor write-back-stores. MWAIT enables a logical processor to enter into an optimized state while waiting for a write-back-store to the address range set up by MONITOR. MONITOR and MWAIT require the use
of general purpose registers for its input. The registers used by MONITOR and MWAIT must be initialized properly; register content is not modified by these instructions.

12.4. SSE3 EXCEPTIONS

SSE3 instructions can generate the same type of memory-access and non-numeric exceptions as other IA-32 architecture instructions. Existing exception handlers generally handle these exceptions without code modification.

FISTTP can generate floating-point exceptions. Some SSE3 instructions can also generate SIMD floating-point exceptions.

SSE3 additions and changes are noted in the following sections. See also: Chapter 11.5., SSE, SSE2, and SSE3 Exceptions.

12.4.1. Device Not Available (DNA) Exceptions

SSE3 will cause a DNA Exception (#NM) if the processor attempts to execute an SSE3 instruction while CR0.TS is set. If CPUID.SSE3 is clear, execution of SSE3 extensions will cause an invalid opcode fault regardless of the state of CR0.TS.

12.4.2. Numeric Error flag and IGNNE#

Most SSE3 instructions ignore CR0.NE (treats it as if it were always set) and the IGNNE# pin. With one exception, all use the vector 19 software exception for error reporting. The exception is FISTTP; it behaves like other x87-FP instructions.

12.4.3. Emulation

The CR0.EM bit used to emulate x87 floating-point instructions cannot be used for emulation of SSE3. If an SSE3 instruction executes when CR0.EM is set, an Invalid Opcode exception (Int 6) is generated instead of a Device Not Available exception (Int 7).

12.5. WRITING APPLICATIONS WITH SSE3 EXTENSIONS

The following sections give guidelines for writing application programs and operating-system code that use SSE3 instructions.

12.5.1. General Guidelines for Using SSE3 Extensions

The following guidelines describe how to maximize the benefits of using SSE3 extensions:

• Ensure that the processor supports SSE3 extensions.
• Ensure that your operating system supports SSE/SSE2/SSE3 extensions. (Operating system support for the SSE extensions implies support for SSE2 extensions, the x87 and SIMD instructions of SSE3 extensions.)

• Ensure your operating system supports MONITOR and MWAIT.

• Employ the optimization and scheduling techniques described in the IA-32 Intel Architecture Optimization Reference Manual (see Section 1.4., Related Literature).

12.5.2. Checking for SSE3 Support

Before an application attempts to use the SIMD subset of SSE3 extensions, the application should follow the steps illustrated in Section 11.6.2., Checking for SSE and SSE2 Support and use the additional step provided below:

7. Check that the processor supports the SIMD and x87 SSE3 extensions. Execute the CPUID instruction with an argument of 1 in the EAX register, and check that the returned data in ECX bit 0 is set to 1.

Checking support for SSE, SSE2 along with SSE3 allows software flexibility to use SSE3. To use FISTTP, software can use the step above to detect support for SSE3.

In the initial implementation of MONITOR and MWAIT, these two instructions are available to ring 0 and conditionally available at ring level greater than 0. Before an application attempts to use the MONITOR and MWAIT instructions, the application should use the following steps:

1. Check that the processor supports MONITOR and MWAIT. Execute the CPUID instruction with an argument of 01H in the EAX register; check that the returned ECX bit 3 is set to 1.

2. If bit 3 of ECX is 1, MONITOR and MWAIT is available at ring 0. To verify MONITOR and MWAIT is supported at ring level greater than 0, application can use a routine similar to Example 12-2 to verify MONITOR and MWAIT is available at application level.

3. Query the minimum and maximum line size that MONITOR will use. The line size can be queried by executing CPUID with 05H in the EAX register.

4. Ensure the memory address range(s) that will be supplied to MONITOR meets memory type requirements.

MONITOR and MWAIT are targeted for system software that supports efficient thread synchronization. See Chapter 12, SSE, SSE2 and SSE3 System Programming in the IA-32 Intel Architecture Software Developer’s Manual, Volume 3 for details.

Verify that SSE3 is supported by using a code sequence like that shown in Example 12-1.
12.5.3. Enable FTZ and DAZ for SIMD Floating-Point Computation

Enabling the FTZ and DAZ flags in the MXCSR register is likely to accelerate SIMD floating-point computation where strict compliance to the IEEE standard 754-1985 is not required. The FTZ flag is available to IA-32 processors that support the SSE; DAZ is available to most IA-32 processors that support SSE2/SSE3.

Software can detect the presence of DAZ, modify the MXCSR register, and save and restore state information by following the techniques discussed in Section 11.6.3. through Section 11.6.6.

12.5.4. Programming SSE3 with SSE and SSE2 Extensions

SIMD instructions in SSE3 extensions are intended to complement the use of SSE/SSE2 in programming SIMD applications. Application software that intends to use SSE3 instructions should also check for the availability of SSE/SSE2 instructions.
The FISTTP instruction in SSE3 is intended to accelerate x87 style programming where performance is limited by frequent floating-point conversion to integers; this happens when the x87 FPU control word is modified frequently. Use of FISTTP can eliminate the need to access the x87 FPU control word.