Willamette** Processor Software Developer's Guide

February, 2000

**Code Name for Future Intel IA-32 Processors

Order Number: 245355-001
World Wide Web: http://developer.intel.com/design/processor/index.htm

TABLE OF CONTENTS

PAGE

PAGE

TABLE OF CONTENTS

PAGE

TABLE OF FIGURES

TABLE OF TABLES

CHAPTER 1 INTRODUCTION TO THE WILLAMETTE PROCESSOR

Willamette is the code name for the next generation of 32 bit Intel® Architecture (IA-32) processors. It is the first member of a new family of processors that are the successors to the Intel P6 family of processors, which includes the Intel Pentium® Pro, Pentium II, and Pentium III processors.

The Willamette processor is based on a new 32-bit micro-architecture from Intel that operates at significantly higher clock speeds and delivers performance levels that are significantly higher than previous generations of IA-32 processors. The Willamette family processors have the following advanced features:

- Higher bandwidth for instruction fetches than the Intel P6 family of processors
- Integer ALU clocked at twice the frequency
  - Reduced latency increases the performance for certain integer operations
- Hyper Pipelined Technology
  - Frequency headroom and performance scalability to continue leadership into the future
  - Deeper pipeline enables world's highest clock rate in desktop PCs
- Streaming SIMD Extension 2
  - 144 New Instructions: Cacheability, SIMD Double-Precision Floating-Point, and SIMD 128-bit integer
  - Enables maximum enjoyment of next generation broadband services such as Interactive Digital TV
- Advance Dynamic Execution
  - Deeper instruction window for out-of-order, speculative execution, improved branch prediction from P6's Dynamic Execution core
- Execution Trace Cache
  - Trace Cache stores pre-decoded micro-ops
- Enhance Floating Point/Multimedia Engine
  - 128-bit FP/Multimedia execution port
  - 128-bit FP load/store port
- Support for Willamette System bus

- — 400 MHz data bus
- — 3.2 Gbytes per second throughput (3x faster than Pentium III processor)

- Advanced Transfer Cache
  - — On die, high bandwidth, low latency 2nd Level Cache
- Compatibility with existing IA-32 applications and operating systems

This chapter provides a high-level overview of the Willamette Processor Architecture. Subsequent chapters provide details of the Streaming SIMD Extension 2 instructions included in the Willamette Processor.

1.1. THE WILLAMETTE PROCESSOR ARCHITECTURE

The Willamette processor architecture enables software developers to leverage new generation of Intel IA-32 architecture to create rich varieties of high performance applications by introducing the following new features:

- Seventy-Six new instructions which include:
  - — New SIMD instructions to operate on high-dynamic-range data including double-precision (64-bit) floating-point values, doubleword integer values, quadword (64-bit) integer data.
  - — New instructions that permit application program to control caches.
- Enhancement to 68 integer SIMD instructions, which worked solely with 64-bit MMX™ registers on Pentium II and Pentium III processors, to work with 128-bit XMM registers in the Willamette processor architecture.

The programming model for the new Willamette processor architecture is similar to but enhanced upon the Intel MMX technology and Streaming SIMD Extensions models with more flexibility. This new architecture allows SIMD computations to be performed on both floating-point and integer data types in the XMM registers (also known as floating-point XMM registers in previously published *Intel Archtecture Software Developer's Manual* for Pentium III processor), or on packed integer data types in the MMX registers.

New SIMD instructions introduced in the Willamette processor architecture include floating-point SIMD instructions, integer SIMD instructions, and conversion of packed data between XMM registers and MMX registers. New Floating-point SIMD instructions allow computations to be performed on packed double-precision floating-point values (two double-precision values per XMM register). Both the single-precision and double-precision floating-point formats and the instructions that operate on them are 100% compatible with IEEE Standard 754 for Binary Floating-Point Arithmetic. New integer SIMD instructions provide flexible and higher dynamic range computational power by supporting arithmetic operations on packed doubleword and quadword data as well as other operations on packed byte, word, doubleword, quadword and double quadword data.

In addition to new SIMD instructions described in the previous paragraph, there are enhancement to 68 integer SIMD instructions, which operated solely on 64-bit MMX registers in the

Pentium II and Pentium III processors, now support operation on 128-bit XMM registers in the Willamette processor architecture. These enhanced integer SIMD instructions allow software developers to have maximum flexibility to implement algorithms by writing SIMD code with either XMM registers or MMX registers.

The Willamette processor architecture features enable software developers to deliver breakthrough levels of performance in multimedia applications ranging from 3-D graphics, video decoding/encoding to speech recognition. The new packed double-precision floating-point instructions enhance performance for applications that require greater range and precision, including scientific and engineering applications and advanced 3-D geometry techniques, such as ray tracing.

To speed up processing and improve cache usage, the Willamette processor architecture offers several new instructions that allow application programmers to control the cacheability of data. These instructions provide the ability to stream data in and out of the registers without disrupting the caches and the ability to prefetch data before it is actually used.

The new architectural features introduced with the Willamette processor architecture do not require new operating system support. This is because the Willamette processor architecture do not introduce new architectural states, and the FXSAVE/FXRSTOR instructions introduced with the Streaming SIMD Extensions is sufficient for saving and restoring the state of the XMM registers, the MMX registers, and the x87 FPU registers during a context switch. The CPUID instruction has been enhanced to allow operating system or applications to test for the existence of the Streaming SIMD Extensions and the Streaming SIMD Extension 2 features.

The Willamette processor architecture is accessible in all IA-32 architecture operating modes. All existing software continues to run correctly, without modification on future IA-32 processors that incorporate the Streaming SIMD Extension 2 instructions. Also, existing software continues to run correctly in the presence of applications that make use of the Streaming SIMD Extension 2 instructions.

1.2. COMPATIBILITY WITH THE P6 FAMILY PROCESSOR ARCHITECTURE

The following features were introduced into the IA-32 architecture within the P6 family of processors and will also be supported in the Willamette processor architecture:

Features added in the Pentium II processors:

- MMX Technology instructions
- SYSENTER/SYSEXIT instructions
- Page attribute table (PAT)
- PSE mode of paging (2^{36} linear address space)

Features added in the Pentium III processors:

- Streaming SIMD Extensions.

- FXSAVE/FXRESTORE instructions.

The following features will be introduced into the IA-32 architecture starting with the Willamette processor and will be supported in all future Willamette family processors:

- Streaming SIMD Extension 2 instructions

The following features are implemented differently in the Willamette processor than in P6 family processors:

- The CPUID instruction has been enhanced to support new features and additional processor information that are not available from the CPUID instruction in the Pentium Pro, Pentium II and Pentium III processors.

1.3. THE WILLAMETTE PROCESSOR MICROARCHITECTURE

The Willamette processor microarchitecture builds upon the dynamic execution techniques pioneered in the P6 family microarchitectures. The Willamette processor microarchitecture extends the P6 family's out-of-order execution model by providing significantly higher instruction fetch bandwidth, increased depth of speculation, and a double pumped ALU. Branch prediction is improved by employing a trace cache to store pre-decoded micro-ops to speed up micro-ops execution and by employing advanced branch-prediction algorithm in conjunction with very deep levels of speculative execution.

1.3.1. Higher Instruction Fetch Bandwidth

P6 family processors are able to fetch, decode, and allocate up to three macro instructions per processor cycle into finer-grained internal units of operation, called micro-ops. The micro-ops can typically execute in a single clock cycle. However the 16-byte fixed length of the instruction decode buffer, the 4-1-1 decoder template, the allocator throughput, and the variable nature of the IA-32 instruction set conspired to reduce the decoder throughput in real-life applications. The Willamette processor microarchitecture is designed to allow the Willamette processor's execution engine to operate at substantially higher clock speeds than P6 family processors.To take advantage of the higher clock speeds provided by the Willamette processor microarchitecture, the Willamette processor employs an execution trace cache as the primary cache to store micro-ops. This allows a high-bandwidth micro-ops instruction stream to be fed into the execution engine. The use of the trace cache in the Willamette processor microarchitecture eliminates the need for a superscalar decoder and removes the instruction decoder from the main execution loop. It also reduces the pipeline bubbles that are caused by branch mispredictions where the front end of the processor has to be redirected to a new decode point.

1.3.2. Increased Depth of Speculation

P6 family processors are able to speculatively execute only up to 40 μops ahead of the retirement pointer. This limitation was imposed by the depth of the Re-Order Buffer (ROB). The Willamette processor microarchitecture's equivalent of the ROB is substantially deeper, allowing significantly deeper speculative execution. This increased speculation depth allows larger or more code loops to be active in the processor and more instruction execution to be hidden behind data cache misses.

1.3.3. Integer And Floating Point Execution Core

P6 family processors are able to execute simple integer operation in a single processor cycle with the more complex floating-point operations taking longer depending upon the operation type. The Willamette processor microarchitecture introduces new technology to the integer execution units that essentially allows very low latency integer operations to be completed at a rate higher than the processor clock frequency. Floating-point operations on the Willamette processor will still take longer to execute than integer operations (depending on the specific floating-point operation) as they do on P6 family processors. However, the floating point performance of the Willamette processor will achieve higher level than the floating point performance of P6 family processors when scaled to the same frequency.

1.3.4. Branch Prediction

As with all long pipeline processors, the better the branch prediction, the better the processor's performance. Willamette processor microarchitecture significantly enhances the branch prediction algorithms originally implemented in the P6 family microarchitecture by effectively combining all currently available prediction schemes.

CHAPTER 2 PROGRAMMING WITH THE STREAMING SIMD EXTENSIONS 2

This chapter provides a general overview of the architectural features that have been added to the IA-32 architecture with the Streaming SIMD Extensions 2.

2.1. STREAMING SIMD EXTENSIONS 2 FEATURE OVERVIEW

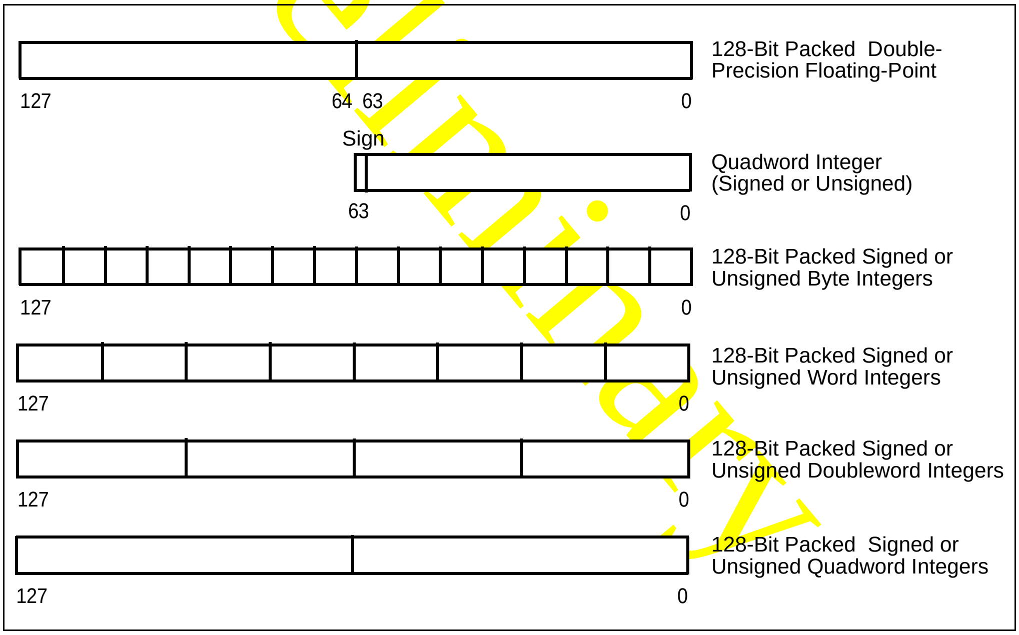
The Streaming SIMD Extensions 2 provide the following new features, while maintaining backward compatibility with all existing IA-32 architecture processors, applications and operating systems.

- New data types:
    - 128-bit packed double-precision floating-point (two double-precision floating-point values packed into a double quadword.
    - 64-bit (quadword) integer (signed and unsigned).
    - 128-bit packed byte integers (signed and unsigned).
    - 128-bit packed word integers (signed and unsigned).
    - 128-bit packed doubleword integers (signed and unsigned).
    - 128-bit packed quadword integers (signed and unsigned).
- New instructions to support the new data types and perform other operations:
    - Packed and scalar double-precision floating-point instructions.
    - Enhanced SIMD integer instructions that support operations on 128-bit operands.
    - Cacheability-control and miscellaneous instructions.
- Modifications to existing IA-32 instructions to support Streaming SIMD Extensions 2 features:
    - Extensions and modifications to the CPUID instruction.
    - Modifications to the RDPMC instruction.

2.2. NEW DATA TYPES

The Streaming SIMD Extensions 2 defines six new data types (see Figure 2-1).

- **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.
- **Quadword integer.** This 64-bit integer can be signed or unsigned. When signed, the sign bit is at bit 63.
- **128-bit packed integers.** These packed integer data types can contain 16 byte integers, 8 word integers, 4 doubleword integers, or 2 quadword integers.



128-Bit Packed Double-Precision Floating-Point
127
64
63
0
Sign
Quadword Integer (Signed or Unsigned)
63
0
128-Bit Packed Signed or Unsigned Byte Integers
127
0
128-Bit Packed Signed or Unsigned Word Integers
127
0
128-Bit Packed Signed or Unsigned Doubleword Integers
127
0
128-Bit Packed Signed or Unsigned Quadword Integers
127
0

Figure 2-1. Streaming SIMD Extensions 2 Data Types

2.3. STREAMING SIMD EXTENSIONS 2 REGISTERS

No new registers are defined with the Streaming SIMD Extensions 2. Streaming SIMD Extensions 2 operations are carried out in the XMM registers, the MMX registers, and/or IA-32 general-purpose registers, as follows.

- **XMM registers.** These eight registers are used to operate on packed or scaler double-precision floating-point data. Scalar operations are operations performed on individual (unpacked) double-precision floating-point values stored in the low order bits of an XMM register. These operations are similar to the operations performed on floating-point data in the x87 FPU registers. The XMM registers are also used to perform operations on 128-bit packed integer data.

- **MMX™ registers.** These eight registers are used for normal MMX technology operations on 64-bit packed integer data and to transfer data to and from the XMM registers.
- **IA-32 general-purpose registers.** MMX™ and XMM registers cannot be used to address memory; addressing is handled through the six general-purpose registers and existing IA-32 addressing modes. The general-purpose registers are also used to hold operands for some of the Streaming SIMD Extensions 2 operations.

The contents of MMX™ and XMM registers are cleared upon reset.

The SIMD floating-point control and status register (MXCSR) controls operations on packed floating-point data in the XMM registers.

2.4. STREAMING SIMD EXTENSIONS 2 INSTRUCTIONS

The Streaming SIMD Extensions 2 instructions are divided into three functional groups

- Packed and scalar double-precision floating-point instructions.
- Enhanced SIMD integer instructions that operate on 128-bit operands.
- Cacheability-control and memory ordering instructions.

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

The packed double-precision floating-point instructions operate similarly to the packed single-precision floating-point instructions, as shown in Figure 2-2.

The scalar double-precision floating-point instructions operate on the low (least significant) quadwords of the two operands as shown in Figure 2-3. Here, the high quadword of the first operand is passed through to the destination.

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

- The MOVUPD instruction supports unaligned accesses.
- Scalar instructions that use an 8-byte memory operand that is not subject to alignment requirements.

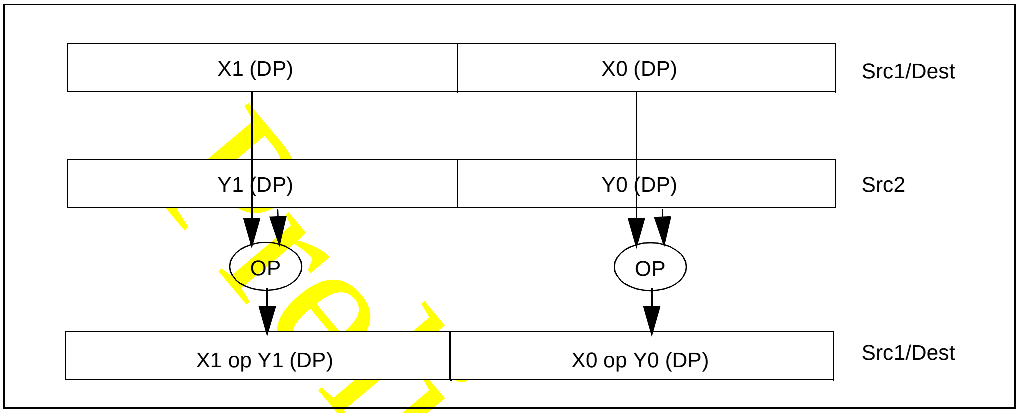


Figure 2-2. Packed Double-Precision Floating-Point Operations

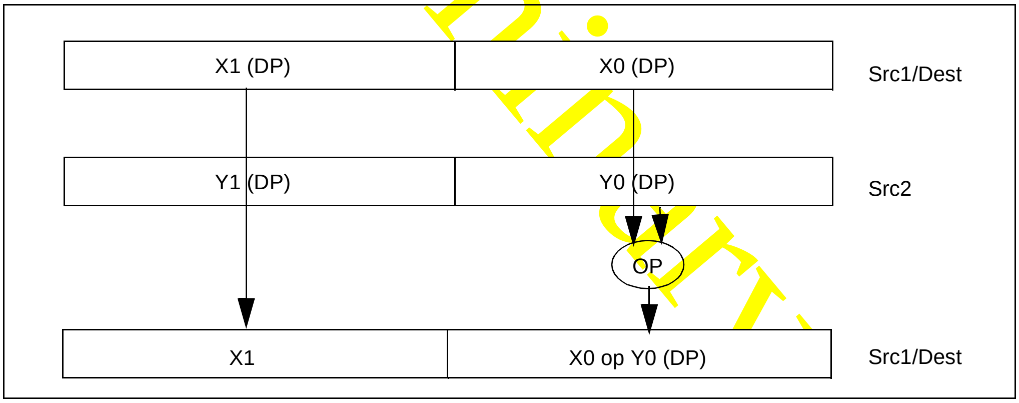


Figure 2-3. Scalar Double-Precision Floating-Point Operations

2.4.1.1. PACKED/SCALAR ADDITION AND SUBTRACTION

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

The ADDSD (Add scalar double-precision floating-point) and SUBSD (Subtract scalar double-precision floating-point) instructions add and subtract, respectively, the low quadword of two double-precision floating-point operands; the high quadword of the source operand is passed through to the destination operand.

2.4.1.2. PACKED/SCALAR MULTIPLICATION AND DIVISION

The MULPD (Multiply packed double-precision floating-point) instruction multiplies two packed double-precision floating-point operands.

The MULSD (Multiply scalar double-precision floating-point) instruction multiplies the low quadwords of two packed double-precision floating-point operands; the high quadword of the source operand is passed through to the destination operand.

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

The DIVSD (Divide scalar double-precision floating-point) instruction divides the low quadwords of two packed double-precision floating-point operands; the high quadword of the source operand is passed through to the destination operand.

2.4.1.3. PACKED/SCALAR SQUARE ROOT

The SQRTPD (Square root packed double-precision floating-point) instruction returns the square roots of a double-precision floating-point operand to the destination operand.

The SQRTSD (Square root scalar double-precision floating-point) instruction returns the square root of the low quadword of the packed double-precision floating-point source operand to the low quadword of the destination operand; the high quadword of the source operand is passed through to the destination operand.

2.4.1.4. PACKED MAXIMUM/MINIMUM

The MAXPD (Maximum packed double-precision floating-point) instruction compares the high quadwords and the low quadwords of two packed double-precision floating-point operands and returns the numerically higher value for each comparison to the destination operand.

The MAXSD (Maximum scalar double-precision floating-point) instruction compares the low (least significant) quadwords of two packed double-precision floating-point operands and returns the numerically higher value for the comparison to the low quadword of the destination operand; the high quadword of the source operand is passed through to the destination operand.

The MINPD (Minimum packed double-precision floating-point) instruction compares the high quadwords and the low quadwords of two packed double-precision floating-point operands and returns the numerically lower value for each comparison to the destination operand.

The MINSD (Minimum scalar double-precision floating-point) instruction compares the low (least significant) quadwords of two packed double-precision floating-point operands and returns the numerically lower value for the comparison to the low quadword of the destination operand; the high quadword of the source operand is passed through to the destination operand.

2.4.1.5. PACKED LOGICAL OPERATIONS

The ANDPD (Bit-wise packed logical AND for double-precision floating-point) instruction returns a bitwise AND between the two operands.

The ANDNPD (Bit-wise packed logical AND NOT for double-precision floating-point) instruction returns a bitwise AND NOT between the two operands.

The ORPD (Bit-wise packed logical OR for double-precision floating-point) instruction returns a bitwise OR between the two operands.

The XORPD (Bit-wise packed logical XOR for double-precision floating-point) instruction returns a bitwise XOR between the two operands.

2.4.1.6. COMPARE OPERATIONS

The CMPPD (Compare packed double-precision floating-point) instruction compares the high quadwords and the low quadwords of two packed double-precision floating-point operands (using an immediate operand as a predicate) and returns a 64-bit mask of all 1s or all 0s as a result for each comparison to the destination operand. The value of the immediate operand allows the selection of any of 12 compare conditions: equal, less than, less than equal, greater than, greater than or equal, unordered, not equal, not less than, not less than or equal, not greater than, not greater than or equal, ordered.

The CMPSD (Compare scalar double-precision floating-point) instruction compares the low quadwords of two packed double-precision floating-point operands (using an immediate operand as a predicate) and returns a 64-bit mask of all 1s or all 0s as a result to the low quadword of the destination operand; the high quadword of the source operand is passed through to the destination operand.

The COMISD (Compare scalar double-precision floating-point ordered and set EFLAGS) instruction compares the low quadwords of two packed double-precision floating-point operands and sets the ZF, PF, and CF flags in the EFLAGS register to show the result. (The OF, SF and AF bits are cleared).

The UCOMISD (Unordered compare scalar double-precision floating-point ordered and set EFLAGS) instruction compares the low quadwords of two packed double-precision floating-point operands and sets the ZF, PF, and CF flags in the EFLAGS register to show the result. (The OF, SF and AF bits are cleared).

2.4.1.7. SHUFFLE OPERATIONS

The SHUFPD (Shuffle packed double-precision floating-point) instruction places either of the two quadwords from first source operand in the low quadword of the destination operand, and places either of the two quadwords from second source operand in the high quadword of the destination operand, (see Figure 2-4).

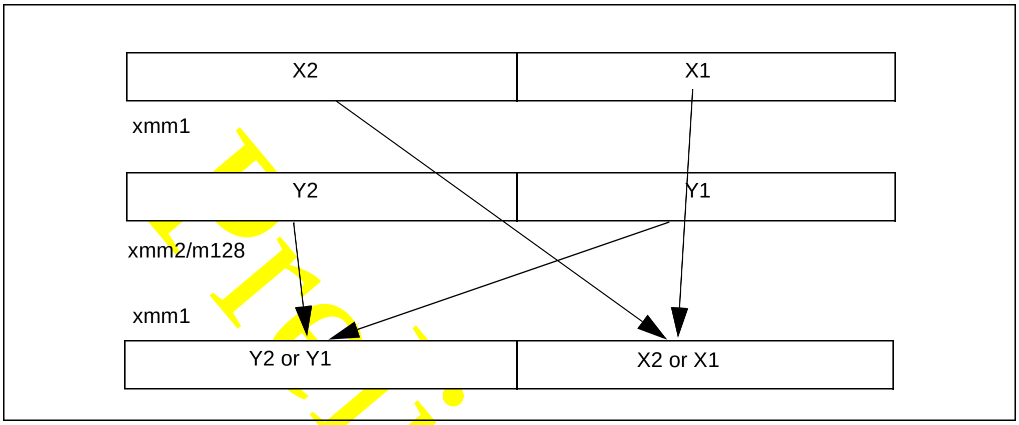


Figure 2-4. Shuffle Operations

The UNPCKHPD (Unpacked high packed double-precision floating-point) instruction performs an interleaved unpack of the high quadwords of the packed double-precision floating-point operands (see Figure 2-5). It ignores the low quadwords of the sources. When unpacking from a memory operand, the full 128-bit operand is accessed from memory but only the high 64 bits are used by the instruction.

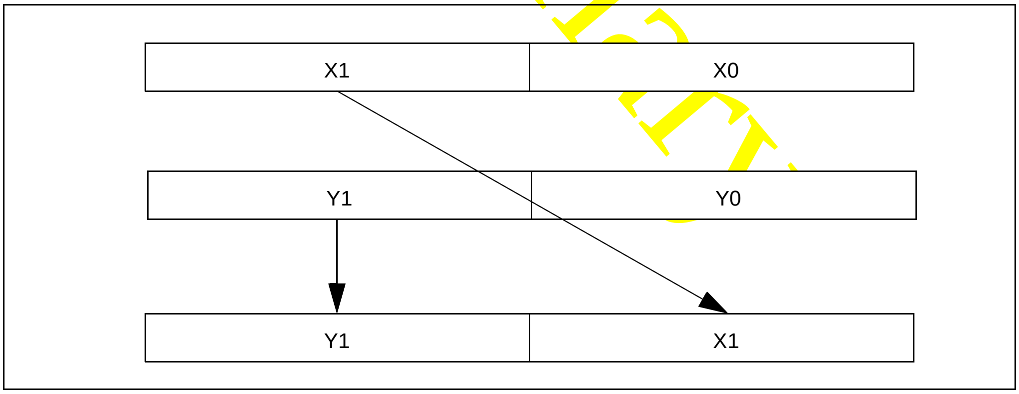


Figure 2-5. Unpack High Operation

The UNPCKLPD (Unpacked low packed double-precision floating-point) instruction performs an interleaved unpack of the low quadwords of the packed double-precision floating-point operands (see Figure 2-6). It ignores the high quadwords of the sources. When unpacking from a memory operand, the full 128-bit operand is accessed from memory but only the low 64 bits are used by the instruction.

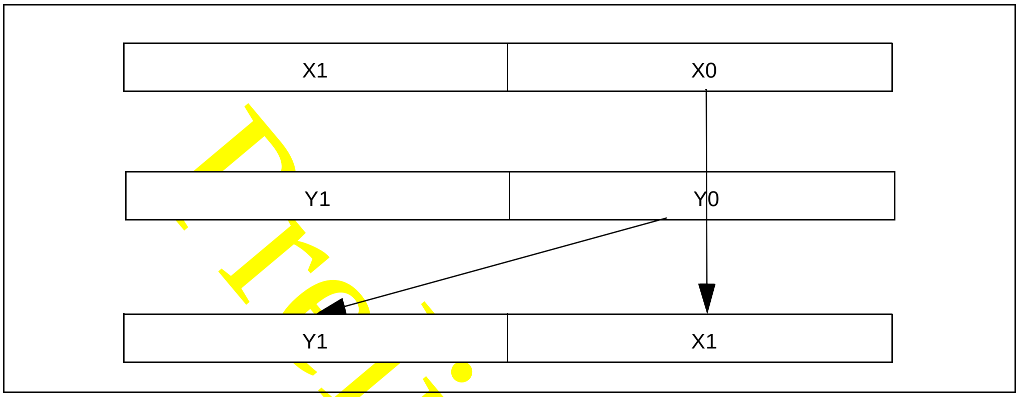


Figure 2-6. Unpack Low Operation

2.4.1.8. CONVERSION INSTRUCTIONS

The Streaming SIMD Extensions 2 conversion instructions (see Figure 2-7) support packed and scalar conversions between:

- Single and double-precision floating-point formats.
- Double-precision floating-point and 32-bit integer formats.

2.4.1.9. CONVERSION BETWEEN DOUBLE-PRECISION FLOATING-POINTS AND 32-BIT INTEGERS

The CVTPD2PI (Convert packed double-precision floating-point to packed 32-bit integer) instruction converts the two packed double-precision floating-point numbers in a 128-bit source operand to two packed 32-bit signed integers, with the result stored in an MMX register. When the conversion is inexact, the rounded value according to the rounding mode in the MXCSR register is returned. The CVTTPD2PI (Convert with truncate packed double-precision floating-point to packed 32-bit integer) instruction is similar to CVTPD2PI except if the conversion is inexact, a truncated result is returned.

The CVTPI2PD (Convert packed 32-bit integer to packed double-precision floating-point) instruction converts two 32-bit signed integers in an MMX register to two double-precision floating-point numbers.

The CVTPD2DQ (Convert packed double-precision floating-point to packed 32-bit integer) instruction converts the two packed double-precision floating-point numbers in a 128-bit source operand to two packed 32-bit signed integers, with the result stored in the low quadword of an XMM register. When the conversion is inexact, the rounded value according to the rounding mode in the MXCSR register is returned. The CVTTPD2DQ (Convert with truncate packed double-precision floating-point to packed 32-bit integer) instruction is similar to CVTPD2DQ except if the conversion is inexact, a truncated result is returned.

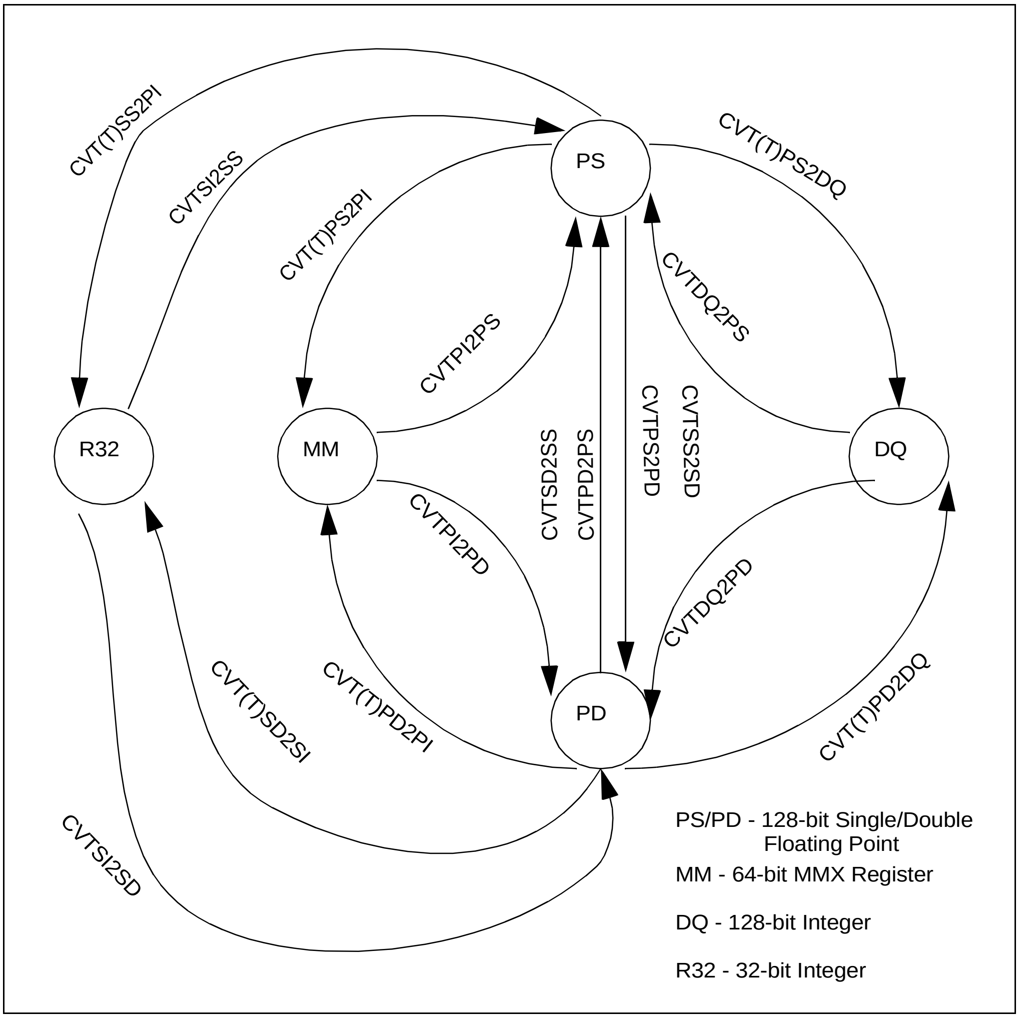


Figure 2-7. Conversion Instructions

The CVTDQ2PD (Convert packed 32-bit integer to packed double-precision floating-point) instruction converts two 32-bit signed integers in an XMM register to two double-precision floating-point numbers.

The CVTSD2SI (Convert scalar double-precision floating-point to a 32-bit integer) instruction converts the low double-precision floating-point number in a 128-bit source operand to a 32-bit signed integer, and stores the result in a general-purpose register. When the conversion is inexact, the rounded value according to the rounding mode in the MXCSR register is returned. The CVTTSD2SI (Convert with truncate scalar double-precision floating-point to scalar 32-bit integer) instruction is similar to CVTSD2SI except if the conversion is inexact, a truncated result is returned.

The CVTSI2SD (Convert 32-bit integer to scalar double-precision floating-point) instruction converts a 32-bit signed integer in a general-purpose register to a double-precision floating-point number, and stores the result in an XMM register.

2.4.1.10. CONVERSION BETWEEN DOUBLE-PRECISION FLOATING-POINTS AND SINGLE-PRECISION FLOATING-POINTS

The CVTPS2PD (Convert packed single-precision floating-point to packed double-precision floating-point) instruction converts the lower two packed single-precision floating-point numbers to two double-precision floating-point numbers in an XMM register.

The CVTPD2PS (Convert packed double-precision floating-point to packed single-precision floating-point) instruction converts the two packed double-precision floating-point numbers to two single-precision floating-point numbers in an XMM register. When the conversion is inexact, the rounded value according to the rounding mode in the MXCSR register is returned.

The CVTSS2SD (Convert scalar single-precision floating-point to scalar double-precision floating-point) instruction converts the lower packed single-precision floating-point number to a double-precision floating-point number in an XMM register.

The CVTSD2SS (Convert scalar double-precision floating-point to scalar single-precision floating-point) instruction converts the lower packed double-precision floating-point number to a single-precision floating-point number in an XMM. When the conversion is inexact, the rounded value according to the rounding mode in the MXCSR register is returned.

2.4.1.11. CONVERSION BETWEEN PACKED SINGLE-PRECISION FLOATING-POINTS AND PACKED INTEGERS

These instructions convert between packed single-precision floating point and packed integer data in XMM registers. These new instructions supplement the conversion instructions introduced in Streaming SIMD Extensions (CVTPI2PS, CVTPS2PI, CVTTPS2PI, CVTSI2SS, CVTSS2SI, CVTTSS2SI)

The CVTPS2DQ (Convert packed single-precision floating-point to packed 32-bit integer) instruction converts four packed single-precision floating-point numbers to four packed 32-bit signed integers, with the source and destination operands are in an XMM register. When the conversion is inexact, the rounded value according to the rounding mode in the MXCSR register is returned. The CVTTPS2DQ (Convert truncate packed single-precision floating-point to packed 32-bit integer) instruction is similar to CVTPS2DQ except if the conversion is inexact, a truncated result is returned.

The CVTDQ2PS (Convert packed 32-bit integer to packed single-precision floating-point) instruction converts four packed 32-bit signed integers to four packed single-precision floating-point numbers, with the source and destination operands are in an XMM register. When the conversion is inexact, the rounded value according to the rounding mode in the MXCSR register is returned.

2.4.1.12. DATA MOVEMENT INSTRUCTIONS

The 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 128 bits of packed data from memory to an XMM register and vice versa, or between XMM registers. The memory address is required to be aligned to a 16-byte boundary; if not, a general protection exception (GP#) is generated.

The MOVHPD (Move high packed double-precision floating-point) instruction transfers 64-bits of packed data from memory to the high quadword of an XMM register and vice versa. The low quadword of the register is left unchanged.

The MOVLPD (Move low packed double-precision floating-point) instruction transfers 64-bits of packed data from memory to the low quadword of an XMM register and vice versa. The high quadword of the register is left unchanged.

The MOVUPD (Move unaligned packed double-precision floating-point) instruction transfers 128 bits of packed data from memory to and XXM register and vice versa, or between XMM registers. No assumption is made for alignment of the memory address.

The MOVMSKPD (Move mask packed double-precision floating-point) instruction transfers the most significant bit of each of the two packed double-precision floating-point numbers to a general-purpose register. This 2-bit value can then be used as a condition to perform branching.

The MOVSD (Move scalar double-precision floating-point) instruction transfers a double-precision floating-point number from memory to an XMM register or vice versa, and between registers.

2.4.2. SIMD Integer Instruction Extensions

The SIMD integer instructions introduced in the MMX technology and the Streaming SIMD Extensions have both been add to and extended, as described in the following sections:

2.4.2.1. NEW SIMD INTEGER INSTRUCTIONS

The Streaming SIMD Extensions 2 adds several new 128-bit instructions to optimize the performance of various applications, such as RSA authentication and RC5 encryption. Where appropriate, a 64-bit version of each of these new instruction is also provided. The 64 bit versions of these new instructions operate on data in MMX registers, and the 128-bit versions of instructions operate on data in the XMM registers. These new instructions are as follows.

The PMULUDQ (Unsigned integer doubleword multiply) instruction performs an unsigned multiply on the low doublewords of each operand, and stores the 64-bit result in the destination operand. Both a 64-bit and a 128-bit version of this instruction is available. The 64-bit version operates on single doubleword integers stored in the low doubleword of each 64-bit source operand, and the quadword result is returned to the 64-bit destination operand. The 128-bit version performs a packed multiply of two pairs of doubleword integers. Here, the doublewords in the packed source operands are stored in bit positions 0 through 31 and 64 through 95 of the

128-bit source operands, and the quadword results are stored in the low and high quadwords of the 128-bit destination operand.

The PADDQ (Packed quadword add) instruction adds 128-bit packed quadword integer operands or 64-bit unpacked quadword integer operands, and returns the 128-bit packed quadword or 64-bit unpacked quadword result to the destination register. This instruction can operate on either unsigned or signed (two's complement notation) integer operands. When an individual result is too large to be represented in 64-bits, the lower 64-bits of the result (wrap-around) are written to the destination operand.

The PSUBQ (Packed quadword subtract) instruction subtracts 128-bit packed quadword integer operands or 64-bit unpacked quadword integer operands, and returns the 128-bit packed quadword or 64-bit unpacked quadword result to the destination register. Like the integer PADDQ instruction, PSUBQ can operate on either unsigned or signed (two's complement notation) integer operands. When an individual result is too large to be represented in 64-bits, the lower 64-bits of the result (wrap-around) are written to the destination operand.

The PSHUFLW/PSHUFHW (Shuffle packed integer words in low or high 64-bits of an XMM register) instruction performs a full shuffle of any source word element within the low or high 64-bits to any result word elements in the low or high 64-bits, using an 8-bit immediate operand to select the shuffle order. The unshuffled words from the source operand are passed through to the result.

The PSHUFD (Shuffle packed integer doublewords in an XMM register) instruction performs a full shuffle of any doubleword field within the 128-bit source to any doubleword field in the 128-bit result, using an 8-bit immediate operand to select the shuffle order.

The PSLLDQ (Shift left logical with byte granularity the contents of an XMM register) instruction shifts the contents of the source operand to the left by the amount of bytes specified by the immediate operand. The empty low-order bytes are cleared (set to zero). If the value specified by the immediate operand is greater than 15, then the destination is set to all zeros.

The PSRLDQ (Shift right logical with byte granularity the contents of an XMM register) instruction shifts the contents of the source operand to the right by the amount of bytes specified by the immediate operand. The empty high-order bytes are cleared (set to zero). If the value specified by the immediate operand is greater than 15, then the destination is set to all zeros.

The PUNPCKHQDQ (Unpack high packed data) 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 low quadwords of the source operands are ignored.

The PUNPCKLQDQ (Unpack low packed data) instruction interleaves the low quadwords of the source operand and the low quadwords of the destination operand and writes them to the destination register. The high quadwords of the source operands are ignored.

Two additional instructions have been added to enable data movement from the MMX registers to the XMM registers.

The MOVMM2DQ (Move integer data from MMX to XMM registers) instruction moves the quadword integer from an MMX source register to an XMM destination register. The high 64 bits of the destination register are cleared to zero.

The MOVDQ2MM(Move integer data from XMM to MMX registers) instruction moves the low quadword integer from an XMM source register to an MMX destination register.

2.4.2.2. EXTENDED SIMD INTEGER INSTRUCTIONS

All of the 64-bit SIMD integer instructions introduced with the MMX technology and the Streaming SIMD Extensions have been extended with the Streaming SIMD Extensions 2 to operate on 128-bit packed integer operands located in the XMM registers. The new 128-bit versions of these instructions follow the same SIMD conventions regarding packed operands as the original 64-bit versions. For example, where a 64-bit version of an instruction operates on 8 packed bytes, the 128-bit version operates on 16 packed bytes.

Since the new 128-bit SIMD integer instructions use the XMM registers instead of the MMX registers, they are not affected by the execution of the x87 FPU. As such the EMMS instruction does not need to be used to preserve MMX register state.

2.4.3. Cacheability Control and Memory Ordering Instructions

The Streaming SIMD Extensions 2 introduces several new instructions to give programs more control over the caching of data and of the loading an storing of data. These new instructions are described in the following sections.

2.4.3.1. CACHE FLUSH

The CLFLUSH (cache line flush) 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 coherency domain.

2.4.3.2. 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 desirable cache temporal data and not cache non-temporal data. The cacheability control instructions enable a program to control caching of non-temporal data to minimize pollution of cached temporal data during non-temporal accesses.

The following four instructions provide hints to the cache hierarchy which enable programs to access non-temporal data with a minimum of cache pollution.

The MASKMOVDQU (Unaligned non-temporal byte mask store of packed integer in an XMM register) instruction stores data from a source XMM register to a memory location specified in the EDI register. The most significant bit in each byte of the second source operand is used as a mask to selectively write the data of the first register on a per-byte basis. The instruction is implicitly weakly-ordered, with all of the characteristics of the WC memory type; successive non-temporal stores may not write memory in program-order, do not write-allocate (i.e. the processor will not fetch the corresponding cache line into the cache hierarchy, prior to performing the store), write combine/collapse, and minimize cache pollution.

The MOVNTDQ (Non-temporal store of packed integer in an XMM register) instruction stores data from a source XMM register to memory. The memory address must be aligned to a 16-byte boundary; if it is not aligned, a general protection exception (GP#) will occur. The instruction is implicitly weakly-ordered, does not write-allocate, and minimizes cache pollution.

The MOVNTPD (Non-temporal store of packed double-precision floating-point) instruction stores data from a source XMM register to memory. The memory address must be aligned to a 16-byte boundary; if it is not aligned, a general protection exception (GP#)will occur. The instruction is implicitly weakly-ordered, does not write-allocate and minimizes cache pollution.

The MOVNTI (Non-temporal store of integer) instruction stores data from a general-purpose register to memory. The data size is always 32 bits, with no alignment restrictions; however unaligned memory accesses may have a performance penalty. The instruction is implicitly weakly-ordered, does not write-allocate, and minimizes cache pollution.

The main difference between a non-temporal store and a regular cacheable store is in the write-allocation policy. The memory type of the region being written to can override the non-temporal hint, leading to the following considerations:

- If a program specifies a non-temporal store to uncacheable memory, then the store behaves like an uncacheable store; the non-temporal hint is ignored and the memory type for the region is retained. Uncacheable as referred to here means that the region being written to has been mapped with either a UC or WP memory type. If the memory region has been mapped as WB, WT or WC, the non-temporal store will implement weakly-ordered (WC) semantic behavior.
- If a program specifies a non-temporal store to cacheable memory, two situations may result:
  - The data is present in the cache hierarchy. Data is evicted from the caches and the new non-temporal data is written to memory (with WC semantics).
  - The data is not present in the cache hierarchy, and the destination region is mapped as WB, WT or WC. The transaction will be weakly ordered, and is subject to all WC memory semantics. The non-temporal store will not write allocate. Different implementations may choose to collapse and combine these stores.
- 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 a fencing operation 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 the line would not be 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. Otherwise 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.

In addition, the processor's execution engine needs to be provided with a continuous flow of data so that it does not become stalled waiting for data. The Streaming SIMD Extensions 2 allow the programmer to prefetch data long before it's final use; the PREFETCH instruction is the same instruction that was defined in Streaming SIMD Extensions. These instructions are not architectural since they do not update any architectural state, and are specific to each implementation. The programmer may have to tune his application for each implementation to take advantage of these instructions. These instructions merely provide a hint to the hardware, and they will not generate exceptions or faults. Excessive use of prefetch instructions may be throttled by the processor.

2.4.3.3. MEMORY ORDERING INSTRUCTIONS

The Streaming SIMD Extensions 2 introduce two new fence instructions (LFENCE and MFENCE) as companions to the SFENCE instruction introduced with the Streaming SIMD Extensions. The LFENCE instruction establishes a memory fence for loads. It guarantees ordering between two loads and prevents speculative loads from passing the memory fence (that is, no speculative loads are allowed until all loads specified before the memory fence have be carried out). This instruction is a companion to the SFENCE instruction.

The MFENCE instruction establishes a memory fence for both loads and stores. It guarantees that all loads and stores specified before the memory fence are globally observable prior to any loads or stores being carried out after the fence. MFENCE combines the functions of the LFENCE and SFENCE instructions.

2.4.3.4. PAUSE

The PAUSE instruction delays execution of the next instruction an implementation specified amount of time. The delay is finite and is predefined for specific Willamette processors.

2.5. FLOATING-POINT TERMINOLOGY AND OPERATIONS

This section describes IEEE Standard 754 floating-point terminology used in describing the double-precision floating-point data type and operations on this data type.

2.5.1. Real Numbers and Floating-Point Formats

This section describes how real numbers are represented in floating-point format in the processor. It also introduces terms such as normalized numbers, denormalized numbers, biased exponents, signed zeros, and NaNs. Readers who are already familiar with floating-point processing techniques and IEEE Standard 754 may wish to skip this section.

2.5.1.1. REAL NUMBER SYSTEM

As shown in Figure 2-8, the real-number system comprises the continuum of real numbers from minus infinity (–∞) to plus infinity (+∞).

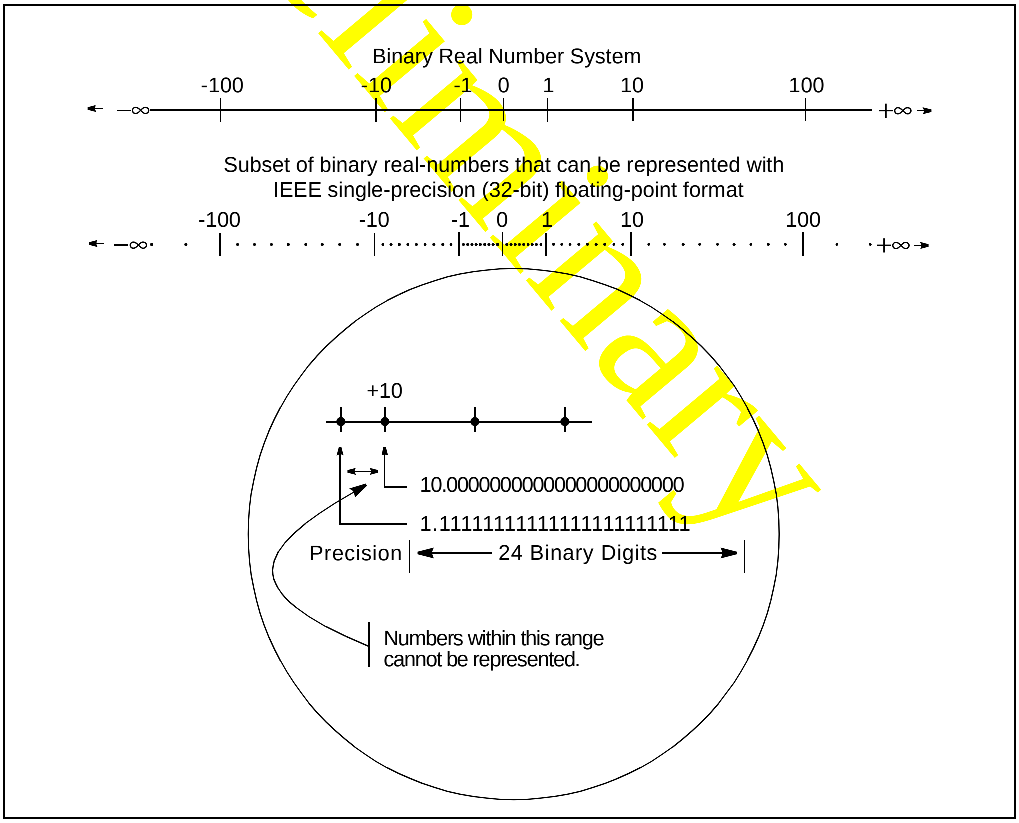

Binary Real Number System
-100
-10
-1
0
1
10
100
–∞
+∞
Subset of binary real-numbers that can be represented with IEEE single-precision (32-bit) floating-point format
-100
-10
-1
0
1
10
100
–∞
+∞
+10
10.0000000000000000000000
1.11111111111111111111111
Precision
24 Binary Digits
Numbers within this range cannot be represented.

Figure 2-8. Binary Real Number System

Because the size and number of registers that any computer can have is limited, only a subset of the real-number continuum can be used in real-number calculations. As shown at the bottom of Figure 2-8, the subset of real numbers that a particular processor supports represents an approximation of the real number system. The range and precision of this real-number subset is determined by the format that the processor uses to represent real numbers.

2.5.1.2. FLOATING-POINT FORMAT

To increase the speed and efficiency of real-number computations, computers typically represent real numbers in a binary floating-point format. In this format, a real number has three parts: a sign, a significand, and an exponent. Figure 2-9 shows the binary floating-point format that Streaming SIMD Extensions 2 data uses. This format conforms to IEEE Standard 754.

The sign is a binary value that indicates whether the number is positive (0) or negative (1). The significand has two parts: a 1-bit binary integer (also referred to as the J-bit) and a binary fraction. The J-bit is often not represented, but instead is an implied value. The exponent is a binary integer that represents the base-2 power that the significand is raised to.

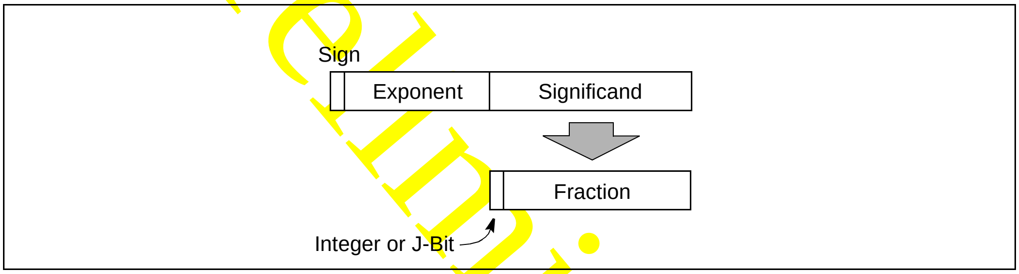


Figure 2-9. Binary Floating-Point Format

Table 2-1 shows how the real number 178.125 (in ordinary decimal format) is stored in the single-precision floating-point format (similar approach for double-precision floating-point). The table lists a progression of real number notations that leads to the format that the processor uses. In this format, the binary real number is normalized and the exponent is biased.

Table 2-1. Real Number Notation

Notation	Value		
Ordinary Decimal	178.125		
Scientific Decimal	$1.78125E_{10}2$		
Scientific Binary	$1.0110010001E_{2}111$		
Scientific Binary (Biased Exponent)	$1.0110010001E_{2}10000110$		
Single Format (Normalized)	Sign	Biased Exponent	Significand
	0	10000110	01100100010000000000000 1. (Implied)

2.5.1.3. NORMALIZED NUMBERS

In most cases, the processor represents real numbers in normalized form. This means that except for zero, the significand is always made up of an integer of 1 and the following fraction:

1.fff...ff

For values less than 1, leading zeros are eliminated. (For each leading zero eliminated, the exponent is decremented by one.)

Representing numbers in normalized form maximizes the number of significant digits that can be accommodated in a significand of a given width. To summarize, a normalized real number consists of a normalized significand that represents a real number between 1 and 2 and an exponent that specifies the number's binary point.

2.5.1.4. BIASED EXPONENT

The processor represents exponents in a biased form. This means that a constant is added to the actual exponent so that the biased exponent is always a positive number. The value of the biasing constant depends on the number of bits available for representing exponents in the floating-point format being used. The biasing constant is chosen so that the smallest normalized number can be reciprocated without overflow.

2.5.1.5. REAL NUMBER AND NON-NUMBER ENCODINGS

A variety of real numbers and special values can be encoded in the processor's floating-point format. These numbers and values are generally divided into the following classes:

- Signed zeros.
- Denormalized finite numbers.
- Normalized finite numbers.
- Signed infinities.
- NaNs.
- Indefinite numbers.

(The term NaN stands for "Not a Number.")

Figure 2-10 shows how the encodings for these numbers and non-numbers fit into the real number continuum. The encodings shown here are for the IEEE double-precision (64-bit) format, where the term "S" indicates the sign bit, "E" the biased exponent, and "F" the fraction. (The exponent values are given in decimal.)

The processor can operate on and/or return any of these values, depending on the type of computation being performed. The following sections describe these number and non-number classes.

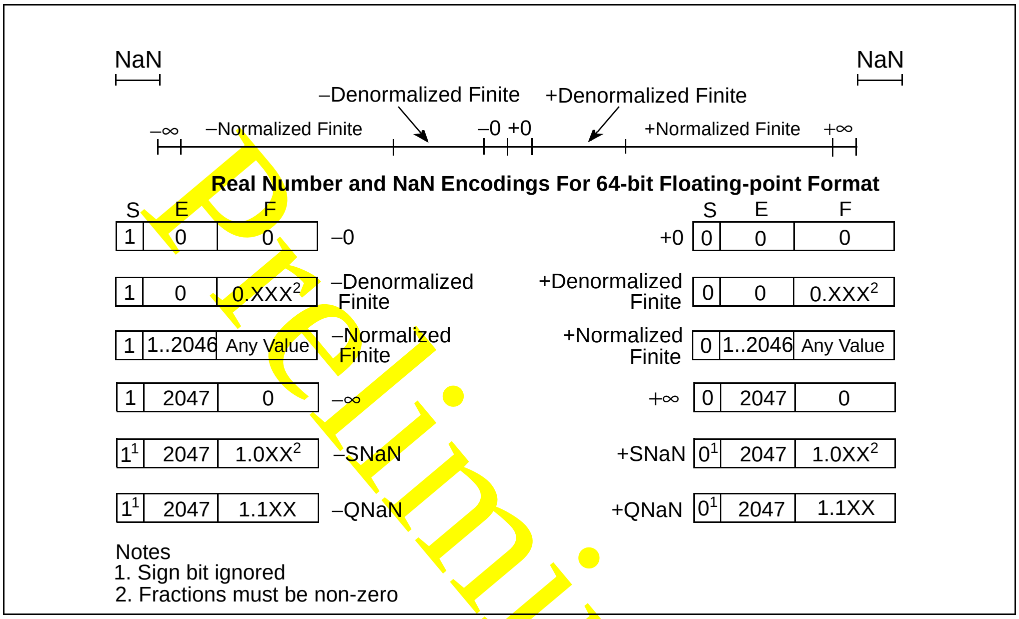


Figure 2-10. Real Numbers and NaNs

2.5.1.6. SIGNED ZEROS

Zero can be represented as a +0 or a −0 depending on the sign bit. Both encodings are equal in value. The sign of a zero result depends on the operation being performed and the rounding mode being used. Signed zeros have been provided to aid in implementing interval arithmetic. The sign of a zero may indicate the direction from which underflow occurred, or it may indicate the sign of an ∞ that has been reciprocated.

2.5.1.7. NORMALIZED AND DENORMALIZED FINITE NUMBERS

Non-zero, finite numbers are divided into two classes: normalized and denormalized. The normalized finite numbers comprise all the non-zero finite values that can be encoded in a normalized real number format between zero and ∞. In the format shown in Figure 2-10., this group of numbers includes all the numbers with biased exponents ranging from 1 to 2046_{10} (unbiased, the exponent range is from -1022_{10} to $+1023_{10}$).

When real numbers become very close to zero, the normalized-number format can no longer be used to represent the numbers. This is because the range of the exponent is not large enough to compensate for shifting the binary point to the right to eliminate leading zeros.

When the biased exponent is zero, smaller numbers can only be represented by making the integer bit (and perhaps other leading bits) of the significand zero. The numbers in this range are called *denormalized* (or *tiny*) numbers. The use of leading zeros with denormalized numbers allows smaller numbers to be represented. However, this denormalization causes a loss of precision (the number of significant bits in the fraction is reduced by the leading zeros).

When performing normalized floating-point computations, a processor normally operates on normalized numbers and produces normalized numbers as results. Denormalized numbers represent an *underflow* condition.

A denormalized number is computed through a technique called gradual underflow. Table 2-2 gives an example of gradual underflow in the denormalization process. Here the single-real format (i.e., as for Streaming SIMD Extensions) is being used, so the minimum exponent (unbiased) is -126_{10}. The true result in this example requires an exponent of -129_{10} in order to have a normalized number. Since -129_{10} is beyond the allowable exponent range, the result is denormalized by inserting leading zeros until the minimum exponent of -126_{10} is reached.

Table 2-2. Denormalization Process

Operation	Sign	Exponent*	Significand
True Result	0	–129	1.01011100000...00
Denormalize	0	–128	0.10101110000...00
Denormalize	0	–127	0.01010111000...00
Denormalize	0	–126	0.00101011100...00
Denormal Result	0	–126	0.00101011100...00

Note
* Expressed as an unbiased, decimal number.

In the extreme case, all the significant bits are shifted out to the right by leading zeros, creating a zero result.

The processor deals with denormal values in the following ways:

- It avoids creating denormals by normalizing numbers whenever possible.
- It provides the floating-point underflow exception to permit programmers to detect cases when denormals are created.
- It provides the floating-point denormal-operand exception to permit procedures or programs to detect when denormals are being used as source operands for computations.

2.5.1.8. SIGNED INFINITIES

The two infinities, $+\infty$ and $-\infty$, represent the maximum positive and negative real numbers, respectively, that can be represented in the floating-point format. Infinity is always represented by a zero significand (fraction and integer bit) and the maximum biased exponent allowed in the specified format (for example, 2047_{10} for the double-real format).

The signs of infinities are observed, and comparisons are possible. Infinities are always interpreted in the affine sense; that is, $-\infty$ is less than any finite number and $+\infty$ is greater than any finite number. Arithmetic on infinities is always exact. Exceptions are generated only when the use of an infinity as a source operand constitutes an invalid operation.

Whereas denormalized numbers represent an underflow condition, the two infinity numbers represent the result of an overflow condition. Here, the normalized result of a computation has a biased exponent greater than the largest allowable exponent for the selected result format.

2.5.1.9. NANS

Since NaNs are non-numbers, they are not part of the real number line. In Figure 2-10, the encoding space for NaNs in the processor floating-point formats is shown above the ends of the real number line. This space includes any value with the maximum allowable biased exponent and a non-zero fraction. (The sign bit is ignored for NaNs.)

IEEE Standard 754 defines two classes of NaN: quiet NaNs (QNaNs) and signaling NaNs (SNaNs). A QNaN is a NaN with the most significant fraction bit set; an SNaN is a NaN with the most significant fraction bit clear. QNaNs are allowed to propagate through most arithmetic operations without signaling an exception. SNaNs generally signal an invalid-operation exception whenever they appear as operands in arithmetic operations. Exceptions are discussed in Section 2.7.

See Section 2.5.2., “Operating on NaNs” for detailed information on how the processor handles NaNs.

2.5.1.10. INDEFINITE

In response to a masked invalid-operation floating-point exceptions, the indefinite value QNAN is produced. The integer indefinite, which can be produced during conversion from double-precision floating-point to 32-bit integer, is defined to be 80000000H (i.e., maximum negative number).

2.5.2. Operating on NaNs

The Streaming SIMD Extensions 2 support the two types of NaNs described in Section 2.5.1.9., “NaNs”: SNaNs and QNaNs. As a general rule, when a QNaN is used in one or more arithmetic floating-point instructions, it is allowed to propagate through a computation. An SNaN on the other hand causes a floating-point invalid-operation exception to be signaled. SNaNs are typically used to trap or invoke an exception handler.

The invalid operation exception has a flag and a mask bit associated with it in the MXCSR register. The mask bit determines how the SNaN value is handled. If the invalid operation mask bit is set, the SNaN is converted to a QNaN by setting the most-significant fraction bit of the value to 1. The result is then stored in the destination operand and the invalid operation flag is set. If the invalid operation mask is clear, an invalid operation fault is signaled and no result is stored in the destination operand.

When a real operation or exception delivers a QNaN result, the value of the result depends on the source operands, as shown in Table 2-3. The exceptions to the behavior described in Table 2-3 are the MINPD, MAXPD, MINSD and MAXSD instructions. If only one source is a NaN for these instructions, the Src2 operand (either NaN or real value) is written to the result; this differs from the behavior for other instructions as defined in Table 2-3, which is to always write the NaN to the result, regardless of which source operand contains the NaN. This approach for MINPD and MAXPD instructions allows NaN data to be screened out of the bounds-checking portion of an algorithm. If instead of this behavior, it is required that the NaN source operand be returned, the min/max functionality can be emulated using a sequence of instructions: comparison followed by AND, ANDN and OR.

Table 2-3. Results of Operations with QNAN operands

Source Operands	NaN Result (Invalid Operation Exception Is Masked)
An SNaN and a QNaN.	Src1 NaN (converted to QNaN if Src1 is an SNaN).
Two SNaNs.	Src1 NaN (converted to QNaN)
Two QNaNs.	Src1 QNaN
An SNaN and a real value.	The SNaN converted into a QNaN.
A QNaN and a real value.	The QNaN source operand.
An SNaN/QNaN value (for instructions which take only one operand such as RCPPS, RCPSS, RSQRTPS, RSQRTSS)	The SNaN converted into a QNaN/the source QNaN.
Neither source operand is a NaN and a floating-point invalid-operation exception is signaled.	The default QNaN *real indefinite*.

In general Src1 and Src2 relate to a Streaming SIMD Extensions 2 as follows:

ADDPD Src1, Src2/m128

Except for the rules given at the beginning of this section for encoding SNaNs and QNaNs, software is free to use the bits in the significand of a NaN for any purpose. Both SNaNs and QNaNs can be encoded to carry and store data, such as diagnostic information.

2.5.3. Streaming SIMD Extensions 2 Data Formats

These sections describe the data formats for the data types introduced in the Streaming SIMD Extensions 2 and the double quadword data type introduced in the Intel Streaming SIMD Extensions.

2.5.3.1. DOUBLE QUADWORD DATA FORMATS

The Intel Streaming SIMD Extensions introduced the 128-bit double quadword data type. For this data type, the bits are numbered 0 through 127, with bit 0 is the least significant bit (LSB), and bit 127 is the most significant bit (MSB). In memory (see Figure 2-11), the bytes of the double quadword data type are located in consecutive memory addresses, using little endian ordering (that is, the bytes with the lower addresses are less significant than the bytes with the higher addresses).

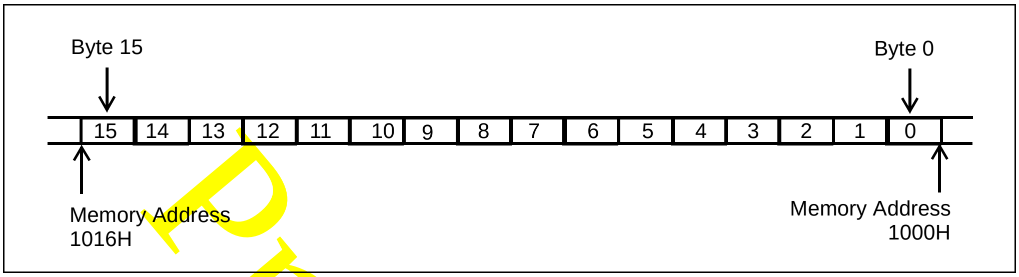


Figure 2-11. Double Quadword Data Type in Memory

2.5.3.2. PACKED DOUBLE-PRECISION FLOATING-POINT FORMAT

The packed double-precision floating-point data type consists of two double-precision floating-point numbers packed into a double quadword, as described in Table 2-4. The individual packed values corresponds directly to the double-precision floating-point format in IEEE Standard 754.

Table 2-4. Double-Precision Floating Point Format

Value	Sign	Exponent	Significand
Low Quadword	63	62...52	51...0
High Quadword	127	126 ... 116	115 ... 64

Table 2-5 gives the precision and range of the double-precision floating-point data type. Only the fraction part of the significand is encoded. The integer is assumed to be 1 for all numbers except 0 and denormalized finite numbers. The exponent of the double-precision floating-point data type is encoded in biased format. The biasing constant is 1023.

Table 2-5. Precision and Range of Double-Precision Floating-Point Data Type

Data Type	Length	Precision (Bits)	Approximate Normalized Range	
			Binary	Decimal
Double Precision	64	53	2^{-1022} to 2^{1024}	2.23×10^{-308} to 1.79×10^{308}

Table 2-6 shows the encodings for all the classes of real numbers (that is, zero, denormalized-finite, normalized-finite, and ∞) and NaNs for the double-precision floating-point data-type. It also gives the format for the real indefinite value, which is a QNaN encoding that is generated by several Streaming SIMD Extensions 2 in response to a masked floating-point invalid-operation exception.

Table 2-6. Real Number and NaN Encodings

Class		Sign	Biased Exponent	Significand	
				Integer[1]	Fraction
Positive	+∞	0	11..11	1	00..00
	+Normals	0 . . 0	11..10 . . 00..01	1 . . 1	11..11 . . 00..00
	+Denormals	0 . . 0	00..00 . . 00..00	0 . . 0	11.11 . . 00..01
	+Zero	0	00..00	0	00..00
Negative	–Zero	1	00..00	0	00..00
	–Denormals	1 . . 1	00..00 . . 00..00	0 . . 0	00..01 . . 11..11
	–Normals	1 . . 1	00..01 . . 11..10	1 . . 1	00..00 . . 11..11
	-∞	1	11..11	1	00..00
NaNs	SNaN	X	11..11	1	0X..XX[2]
	QNaN	X	11..11	1	1X..XX
	Real Indefinite (QNaN)	1	11..11	1	10..00
	Double-Real		⟵ 11 Bits ⟶		⟵ 52 Bits ⟶

When storing packed double-precision floating-point values in memory, each of the packed values are stored in 8 consecutive bytes in memory. The 128-bit access mode is used for 128-bit memory accesses, 128-bit transfers between XMM registers, and all logical, unpack and arithmetic instructions.The 64-bit access mode is used for 64-bit memory access, 64-bit transfers between XMM registers, and all arithmetic instructions.

2.5.3.3. IEEE COMPLIANCE

The double-precision floating-point data type and operations performed on this data type with the Streaming SIMD Extensions 2 comply with IEEE Standard 754, except for the flush to zero mode. IEEE Standard 754 compliance includes support for double-precision signed infinities, QNaNs, SNaNs, integer indefinite, signed zeros, denormals, masked and unmasked exceptions (assuming appropriate operating support and exception handler support for the latter). Double precision floating-point values are represented identically both internally and in memory in the format shown in Table 2-4.

This is a change from x87 FPU which internally represents all numbers in 80-bit extended double-precision floating-point format. This change implies that x87 FPU libraries that are rewritten to use Streaming SIMD Extensions 2 may not produce results that are identical to the those produced by x87 FPU instructions (i.e., double rounding can occur with x87 FPU underflows that are followed by a store to memory).

2.5.3.4. 128-BIT PACKED INTEGER FORMAT

Integer values can be packed into a double quadword in any of four formats: 16 packed byte integers, 8 packed word integers, 4 packed doubleword integers, and 2 packed quadword integers.

The same format is used to pack integers into XMM registers as is used in memory. They have four data access modes: 128-bit accesses, 64-bit accesses, 32-bit accesses and 16-bit accesses.

- The 128-bit access mode is used for 128-bit memory access (aligned), 128-bit transfer to/from an XMM register (aligned and unaligned), some unpack instructions, all pack instructions, and all logical and arithmetic instructions.
- The 64-bit access mode is used for 64-bit memory access, 64-bit transfer to/from XMM registers, some unpack instructions, all pack instructions, and all logical and arithmetic instructions.
- The 32-bit access mode is used for 32-bit memory access, 32-bit transfer from a general-purpose register, and some unpack instructions.
- The 16-bit access mode is used for insertion/extraction of one 16-bit word into a 64-bit MMX or a 128-bit XMM register.

2.5.4. MXCSR Register

The MXCSR register is used to enable masked/unmasked floating-point exception handling, to set rounding modes, to set flush-to-zero mode, and to view status flags. The contents of this register can be loaded with the LDMXCSR and FXRSTOR instructions and stored in memory with the STMXCSR and FXSAVE instructions. Figure 2-12 shows the format and encoding of the fields in the MXCSR register.

31 16	15	14 13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	FZ	RC	PM	UM	OM	ZM	DM	IM	Rsv	PE	UE	OE	ZE	DE	IE

Figure 2-12. MXCRS Register Format

Bits 5-0 indicate whether an floating-point exception has been detected. They are “sticky” flags, and can be cleared by using the LDMXCSR and FXRSTOR instructions to write zeroes to these fields. If a LDMXCSR/FXRSTOR instruction clears a mask bit and sets the corresponding exception flag bit, an exception will not be immediately generated. The exception will occur only upon the next Streaming SIMD Extensions or Streaming SIMD Extensions 2 instruction to cause this type of exception. Streaming SIMD Extensions and Streaming SIMD Extensions 2 use only one exception flag for each exception. There is no provision for individual exception reporting within a packed data type. In situations where multiple identical exceptions occur within the same instruction, the associated exception flag is updated and indicates that at least one of these conditions happened. These flags are cleared upon reset.

Bits 12-7 configure floating-point exception masking; an exception type is masked if the corresponding bit is set and it is unmasked if the bit is clear. These bits are set upon a processor reset, meaning that all floating-point exceptions are masked.

Bits 14-13 encode the rounding-control, which provides for the common round-to-nearest mode, as well as directed rounding and chop (refer to Section 2.5.5., "Rounding Control Field"). The rounding-control is set to round to nearest upon reset.

Bit 15 (FZ) is used to turn on the flush to zero mode (refer to Section 2.5.6., "Flush to Zero"). This bit is cleared upon reset, disabling the flush to zero mode.

The other bits of the MXCSR register (bit 6 and bits 16 through 31) are defined as reserved and cleared; 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#).

2.5.5. Rounding Control Field

The rounding-control (RC) field of the MXCSR register (bits 13 and 14) controls how the results of floating-point instructions are rounded. Four rounding modes are supported: round to nearest, round up, round down, and round toward zero (see Table 2-7). Round to nearest is the default rounding mode and is suitable for most applications. It provides the most accurate and statistically unbiased estimate of the true result.

Table 2-7. Rounding Control Field (RC)

Rounding Mode	RC Field Setting	Description
Round to nearest (even)	00B	Rounded result is the closest to the infinitely precise result. If two values are equally close, the result is the even value (that is, the one with the least-significant bit of zero).
Round down (toward $-\infty$)	01B	Rounded result is close to but no greater than the infinitely precise result.
Round up (toward $+\infty$)	10B	Rounded result is close to but no less than he infinitely precise result.
Round toward zero (Truncate)	11B	Rounded result is close to but no greater in absolute value than the infinitely precise result.

The round up and round down modes are termed *directed rounding* and can be used to implement interval arithmetic. Interval arithmetic is used to determine upper and lower bounds for the true result of a multistep computation, when the intermediate results of the computation are subject to rounding.

The round toward zero mode (sometimes called the "chop" mode) is commonly used when performing integer arithmetic with the processor.

Whenever possible, the processor produces an infinitely precise result. However, it is often the case that the infinitely precise result of an arithmetic operation cannot be encoded exactly in the format of the destination operand. For example, the following single-precision value (*a*) has a 24-bit fraction (similar concept for double-precision values). The least-significant bit of this fraction (the underlined bit) cannot be encoded exactly in the single-real format (which has only a 23-bit fraction):

(*a*) 1.0001 0000 1000 0011 1001 011$\underline{1}$$E_2$ 101

To round this result (*a*), the processor first selects two representable fractions *b* and *c* that most closely bracket *a* in value ($b < a < c$).

(*b*) 1.0001 0000 1000 0011 1001 011E_2 101

(*c*) 1.0001 0000 1000 0011 1001 100E_2 101

The processor then sets the result to *b* or to *c* according to the rounding mode selected in the RC field. Rounding introduces an error in a result that is less than one unit in the last place to which the result is rounded (for round down, round up and round toward zero); for round to nearest the error is less than or equal to 1/2 unit in the last place to which the result is rounded.

The rounded result is called the inexact result. When the processor produces an inexact result, the floating-point precision (inexact) flag (PE) is set in the MXCSR register.

When the infinitely precise result is between the largest positive finite value allowed in a particular format and +∞, the processor rounds the result as shown in Table 2-8.

Table 2-8. Rounding of Positive Numbers

Rounding Mode	Result
Rounding to nearest (even)	+∞
Rounding down (toward –∞)	Maximum, positive finite value
Rounding up (toward +∞)	+∞
Rounding toward zero (Truncate)	Maximum, positive finite value

When the infinitely precise result is between the largest negative finite value allowed in a particular format and –∞, the processor rounds the result as shown in Table 2-9.

Table 2-9. Rounding of Negative Numbers

Rounding Mode	Result
Rounding to nearest (even)	-∞
Rounding toward zero (Truncate)	Maximum, negative finite value
Rounding up (toward +∞)	Maximum, negative finite value
Rounding down) (toward –∞)	-∞

The rounding modes have no effect on comparison operations, operations that produce exact results, or operations that produce NaN results.

2.5.6. Flush to Zero

Turning on the flush-to-zero mode has the following effects during underflow situations:

- Zero results are returned with the sign of the true result.
- Precision and underflow exception flags are set.

The IEEE mandated masked response to underflow is to deliver the denormalized result (i.e., through the gradual underflow procedure); consequently, the flush to zero mode is not compatible with IEEE Standard 754. It is provided primarily for performance reasons. At the cost of a slight precision loss, faster execution can be achieved for applications where underflow are common. Underflow for flush to zero is defined to occur when the exponent for a computed result, prior to denormalization scaling, falls in the denormal range; this is regardless of whether a loss of accuracy has occurred. Unmasking the underflow exception takes precedence over flush-to-zero mode; this means that an exception handler will be invoked for an XMM instruction that generates an underflow condition while this exception is unmasked, regardless of whether flush to zero is enabled.

2.6. WRITING PROGRAMS USING THE STREAMING SIMD EXTENSIONS 2

The following sections give some guidelines for writing application programs and operating-system code to use the new data types and instructions introduced with the Streaming SIMD Extensions 2.

2.6.1. Using the CPUID Instruction to Detecting the Existence of the Streaming SIMD Extensions 2

The Streaming SIMD Extensions and Streaming SIMD Extensions 2 can be divided into four categories:

- Single and double-precision packed/scalar floating-point.

- New and enhanced SIMD integer instructions.
- State management instructions (i.e., LDMXCSR/STMXCSR).
- Cacheability control instructions, further subdivided as:
  - Streaming stores for both packed floating-point (MOVNTPS and MOVNTPD) and integer MMX (MASKMOVQ, MASKMOVDQU, MOVNTQ and MOVNTDQ) and general purpose integer data (MOVNTI)
  - CLFLUSH, PREFETCH, SFENCE, LFENCE and MFENCE that are not constrained to work with any specific data type.

To use these features, the following conditions must exist, otherwise an invalid opcode exception (#UD) is generated.

- CR0.EM = 0 (emulation disabled)
- CR4.OSFXSR = 1 (operating system supports saving Streaming SIMD Extensions and Streaming SIMD Extensions 2 state during context switches)
- CPUID.WNI = 1 (processor supports Streaming SIMD Extensions 2)

An application can verify that Streaming SIMD Extensions 2 are supported by performing the following code sequence:

```
boolean willamette_new_instructions_work = TRUE;
try {
        IssueWillametteNewInstruction();
        // Use XORPD
except (UNWIND) {
        // if we get here, Streaming SIMD Extensions 2 doesn't work
        willamette_new_instructions_work = FALSE;
}
```

Similarly, an application can verify support for unmasked SIMD floating-point exceptions by performing the following sequence:

```
boolean willamette_new_instruction_unmasked_works = TRUE;
try {
        IssueUnMask();
        // Unmask divide by zero and compute 1/0 using DIVPD
except (UNWIND) {
        // if we get here, unmasked exceptions don't work
        willamette_new_instruction_unmasked_works = FALSE;
}
```

To verify support for the new and enhanced SIMD integer instructions and the cacheability control instructions, the application needs to perform the following code sequence:

```
boolean willamette_new_instructions_work = TRUE;
try {
        IssueWillametteNewInstruction();
        // Use PADDQ
except (UNWIND) {
        // if we get here, Streaming SIMD Extensions 2 doesn't work
        willamette_new_instructions_work = FALSE;
}
```

It is important that the instructions indicated be used within these try-except sequences; use of another instruction may not correctly identify whether support for the enhancements is available.

Similar to the Streaming SIMD Extensions and Packed Double-Precision floating-point instructions, the new and enhanced SIMD integer instructions will generate an invalid opcode exception (#UD) if CR0.EM is set.

To verify support for the PREFETCH and SFENCE instructions, the application needs only to check that either CPUID.XMM is set to 1 or CPUID.WNI is set to 1. To verify support for CLFLUSH, LFENCE and MFENCE, the applications needs to check that CPUID.WNI is set to 1. These five instructions are not affected by CR0.EM or CR4.OSFXSR.

2.6.2. Updating Existing MMX Technology Routines Using 128-bit Integer Enhancements

The Streaming SIMD Extensions 2 extend all of the 64-bit SIMD integer instructions introduced with the MMX technology to operate on 128-bit SIMD integers located in the XMM registers. The extended 128-bit SIMD integer instructions operate the same as the 64-bit SIMD integer instructions, which simplifies porting of current MMX technology applications to wider packed operations in the XMM registers. However, there are few additional considerations:

1. The 128-bit SIMD integer instructions operate on the XMM registers. Thus existing MMX technology code will need to be recompiled to take advantage of the wider the 128-bit SIMD integer instructions.
2. 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 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 will result in a General Protection exception (#GP).
3. 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, PSHUFD.
4. Use of the 64-bit shift by bit instructions (PSRLQ, PSLLQ) are extended to 128 bits in two ways:

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

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

2.6.3. Interaction of Streaming SIMD Extensions 2 and x87-FPU and MMX Instructions

Since the 128-bit SIMD integer instructions use the XMM registers (with a few exceptions), there are no restrictions on the simultaneous use of x87 FPU or MMX instructions. For instance, the EMMS instruction is not necessary when integrating a floating-point Streaming SIMD Extensions or Streaming SIMD Extensions 2 module with existing MMX technology modules or existing x87 FPU modules. Streaming SIMD Extensions and Streaming SIMD Extensions 2 also do not affect, nor are affected by, the x87 FPU tag word, control word, or status word, or the floating-point exception state.

The instructions CVTPS2PI, CVTTPS2PI, CVTPI2PS, CVTPD2PI, CVTTPD2PI, CVTPI2PD, MOVDQ2Q and MOVQ2DQ use MMX registers and are subject to the same restrictions as the current MMX technology instructions on the simultaneous use of the x87 FPU, which include:

- Transition from x87 FPU to MMX technology (TOS=0, FP valid bits set to all valid).
- Use of EMMS for transition from MMX technology code to x87 FPU code.

Note that the EMMS instruction still needs to be executed when switching from MMX technology code to x87 FPU code.

2.6.3.1. FXSAVE/FXRSTOR REPLACES USE OF FSAVE/FRSTOR

The FSAVE instruction does not save any of the new state associated with Streaming SIMD Extensions and Streaming SIMD Extensions 2. The use of the FSAVE and FRSTOR instructions should be replaced with the FXSAVE and FXRSTOR instructions, which save and restore the full XMM register state along with the x87 FPU (and MMX) register state. Exception and interrupt handlers which use 64-bit SIMD integer operations or x87 FPU operations are instance where the FSAVE and FRSTOR instructions should be replaced with the FXSAVE and FXRSTOR instructions.

2.6.3.2. INTERACTION OF PACKED OR SCALAR FLOATING-POINT INSTRUCTIONS WITH THE X87 FPU AND MMX INSTRUCTIONS

The XMM registers are separate from the x87 FPU and MMX registers. An application can use the packed or scalar floating-point instructions of Streaming SIMD Extensions and Streaming SIMD Extensions 2 with either 64-bit SIMD integer instructions or x87 FPU instructions simultaneously, without any penalty. An application can use x87 FPU for operations that need extended precision arithmetic, or for accessing any of the x87 FPU trigonometric instructions.

2.6.3.3. INTERMIXING OF PACKED AND SCALAR FLOATING-POINT INSTRUCTIONS AND DATA

This section summarizes the various cases where instructions of one type use data of another type (for instructions that read/write the packed/scalar XMM registers). Note that there is a latency penalty associated with these types of data intermixing, which can lead to significant performance degradations if this intermixing occurs within an inner code loop.

Single-precision floating-point instruction operates on double-precision floating-point or 128-bit SIMD integer data: The instruction operates on the double-precision floating-point source data in the same format as seen in memory; (i.e., each 64-bit double-precision floating-point or 128-bit integer source operand is viewed as two 32-bit single-precision operands). Optimal performance for this type of data interaction will not be guaranteed for a given implementation.

Double-precision floating-point instruction operates on single-precision floating-point or 128-bit SIMD integer data: The instruction operates on the single-precision floating-point source data in the same format as seen in memory; (i.e., two 32-bit single-precision or 128-bit integer source operands are viewed as a single 64-bit double-precision operand). Optimal performance for this type of data interaction will not be guaranteed for a given implementation.

128-bit SIMD integer instruction operates on single-precision floating-point data: The instruction operates on the single-precision source data in the same format as seen in memory; (i.e., four 32-bit single-precision floating-point source operands are viewed as a single 128-bit integer operand. Optimal performance for this type of data interaction will not be guaranteed for a given implementation.

128-bit SIMD integer instruction operates on double-precision floating-point data: The instruction operates on the double-precision floating-point source data in the same format as seen in memory; (i.e., each two 64-bit double-precision floating-point source operand is viewed as a 128-bit SIMD integer operands). Optimal performance for this type of data interaction will not be guaranteed for a given implementation.

While it is not usual to perform double-precision floating-point operations on single-precision floating-point data (or vice-versa), there are some cases which might arise in common usage:

1. The XORPS, XORSS, XORPD, and XORSD instructions might be used with the same register for both source operands, to clear the contents of the register prior to a subsequent operation. If the initial register contents are of a different type than the XOR, a latency penalty will be incurred.

2. Data movement instructions (including the MOVAPS, MOVUPS, MOVSS, MOVAPD, MOVUPD, MOVSD, MOVHPS, MOVLPS, MOVHPD, MOVLPD, MOVNTPS and MOVNTPD instructions) perform a typed store; if the register accessed contains data of a different type than that of the instruction, a latency penalty will be incurred.

3. The MOVHPS, MOVLPS, MOVHPD and MOVLPD instructions load half of an XMM register with data from memory and leave the other half intact. This operation performs a merging of data in the register with data loaded from memory, each of which may be of a different type (single- or double-precision floating-point); as a result, if the initial register contents are of a different type than the XOR, a latency penalty will be incurred.

2.6.4. Caller-Save Requirement for Function Calls

The XMM registers can operate on three different data types (packed single-precision floating-point, packed double precision floating-point, packed integer data). A caller-save convention for function-calls is recommended. An application which first uses 128-bit packed integer operations and then calls another function must assume that all the 128-bit registers will be overwritten by either packed single-precision floating-point or packed double-precision floating-point operations used within the called function. This situation requires the calling application to save (i.e., push onto the stack) the contents of XMM registers that will be needed upon returning from the function.

The use of a callee-save convention for function calls is strongly discouraged, due to a resulting performance impact. Callee-save means the calling function can leave values in registers and expect the values to remain intact upon returning from a function. The called function is thus expected to save those registers it will modify and restores them prior to returning. If the called function does not use load/store instructions of the same data type (x87 FPU, 64-bit/128-bit packed integer, single- and double-precision floating-point) as the values in the registers, there will be a performance impact, as indicated in Section 2.6.3.3.. Streaming SIMD Extensions 2 do not provide any method of determining which type of data is in the register, to select the correctly typed load/store instruction.

As a result, it is strongly recommended that programs follow a caller-save convention, whereby any registers whose contents must survive intact across a function call are stored prior to executing the call. Upon returning from the call, the calling program is responsible for restoring/loading the appropriate registers.

2.6.5. Cacheability Hint Instructions

The Streaming SIMD Extensions 2 cacheability control instructions enable the programmer to control caching and prefetching of data. When correctly used, these instructions can significantly improve application performance.

The PREFETCH instruction (introduced in the Streaming SIMD Extensions) can minimize the latency of data access in performance-critical sections of application code by allowing data to be fetched in advance of actual usage.

The non-temporal store instructions (MOVNTPD, MOVNTDQ, MASKMOVDQU) minimize cache pollution while writing data. The main difference between a non-temporal store and a regular cacheable store is in the write-allocation policy. The memory type can override the non-temporal hint, leading to the following scenarios:

- If a program specifies a non-temporal store to uncacheable memory, then the store behaves like an uncacheable store; the non-temporal hint is ignored and the memory type retained.
- If a program specifies a non-temporal store to cacheable memory. Two cases may result:
    - If the data is present in the cache hierarchy, cache coherency will be maintained and the existing memory type attributes are retained.

- If the data is not present in the cache hierarchy, the memory type visible on the bus will remain unchanged and the transaction will be weakly ordered. Consequently, the program is responsible for maintaining coherency. Non-temporal stores will not write allocate (i.e. the processor will not fetch the corresponding cache line into the cache hierarchy, prior to performing the store). Different implementations may choose to collapse and combine these stores.

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 it to see. Some common usage models which may be affected in this way by weakly-ordered stores are:

- Library functions, which use weakly ordered memory to write results.
- Compiler-generated code, which also benefit from writing 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 instruction should be used to ensure ordering between routines that produce weakly-ordered data and routines that consume this data. The SFENCE instruction provides a performance-efficient way to ensure ordering, by guaranteeing that every store instruction that precedes the SFENCE instruction in program order is globally visible before any store instruction which follows the fence.

2.6.6. Branching on Streaming SIMD Extensions 2 Arithmetic Operations

There are no condition codes in Streaming SIMD Extensions and Streaming SIMD Extensions 2 state. A packed-data comparison instruction generates a mask which can then be transferred to an integer register. The following code sequence is an example of how to perform a conditional branch, based on the result of an Streaming SIMD Extensions 2 arithmetic operation.

```
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 scalar comparison. A conditional branch can then be scheduled immediately following the COMISD/UCOMISD instruction.

2.6.7. Saving the Streaming SIMD Extensions and Streaming SIMD Extensions 2 State

An application needs to identify the nature of the multitasking operating system on which it runs. Each task retains its own state which must be saved when a task switch occurs. The processor state (context) consists of the integer registers, x87 FPU and MMX registers, and XMM registers. The STMXCSR and FXSAVE instructions store Streaming SIMD Extensions and Streaming SIMD Extensions 2 state in memory for use by exception handlers and other system and application software. The STMXCSR instruction saves the contents of the MXCSR register. The FXSAVE instruction saves the x87 FPU state (status, control, tag, instruction pointer, data pointer, opcode and stack registers) and Streaming SIMD Extensions and Streaming SIMD Extensions 2 state (status/control and data registers). An application needs to verify that the processor supports FXSAVE prior to using this instruction. For a processor that implements FXSAVE but not Streaming SIMD Extensions 2, this can be done by checking the CPUID.FXSR bit; for a processor that does implement Streaming SIMD Extensions 2, the approach described in Section 2.6.1. should be used.

The operating systems can be classified into two types:

- Cooperative multitasking operating systems
- Preemptive multitasking operating systems

2.6.7.1. COOPERATIVE MULTITASKING OPERATING SYSTEM ENVIRONMENT

A cooperative multitasking operating system does not save the x87 FPU and MMX technology state or the Streaming SIMD Extensions and Streaming SIMD Extensions 2 state when performing a context switch. Therefore, the application needs to save the relevant state before relinquishing direct or indirect control to the operating system.

2.6.7.2. PREEMPTIVE MULTITASKING OPERATING SYSTEM ENVIRONMENT

A preemptive multitasking operating system saves the x87 FPU and MMX technology state and the Streaming SIMD Extensions and Streaming SIMD Extensions 2 state when performing a context switch. Therefore, the application does not have to save or restore Streaming SIMD Extensions and Streaming SIMD Extensions 2 state.

2.6.8. Initialization of Streaming SIMD Extensions and Streaming SIMD Extensions 2 Technology

A hardware reset affects Streaming SIMD Extensions and Streaming SIMD Extensions 2 floating-point state as follows (see Table 2-10: all exceptions are masked, all exception flags are cleared, the rounding control is set to round-nearest, and the flush-to-zero mode is disabled. If the processor is reset by asserting the INIT# pin, the Streaming SIMD Extensions and Streaming SIMD Extensions 2 state is not changed.

Table 2-10. Streaming SIMD Extensions State Following Reset or INIT

Streaming SIMD Extensions and Streaming SIMD Extensions 2 Register	Power-Up or Reset	INIT
XMM0 through XMM7	+0.0	Unchanged
MXCSR	1F80H	Unchanged

2.6.9. Interfacing with Streaming SIMD Extensions 2 Procedures and Functions

The Streaming SIMD Extensions 2 allows direct access to all of the 128-bit XMM registers. This means that all existing interface conventions that apply to the use of the general-purpose registers (EAX, EBX, etc.) will also apply to Streaming SIMD Extensions 2 register usage.

2.7. HANDLING EXCEPTIONS IN STREAMING SIMD EXTENSIONS 2 OPERATIONS

Similar to packed/scalar Streaming SIMD Extensions, Streaming SIMD Extensions 2 generate two kinds of exceptions:

- Non-numeric exceptions
- Numeric exceptions

Streaming SIMD Extensions 2 can generate the same type of memory access exceptions as other IA-32 Architecture instructions. Some examples are page fault, segment not present, and limit violations. Existing exception handlers can handle these types of exceptions without any code modification. The PREFETCH hints will not generate any kind of exception and instead the instruction will be ignored.

The Streaming SIMD Extensions 2 generate the same six floating-point exceptions for SIMD double-precision floating-point operations that x87 FPU instructions generate. All SIMD floating-point exceptions are reported independently of x87 FPU floating-point exceptions. Independent masking and unmasking of SIMD floating-point exceptions is achieved by setting and clearing specific bits in the MXCSR register.

The application must ensure that the operating system can support unmasked SIMD floating-point exceptions before unmasking them, as described in Section 2.6.1., “Using the CPUID Instruction to Detecting the Existence of the Streaming SIMD Extensions 2”. If an application unmasks exceptions using either FXRSTOR or LDMXCSR without the required operating system support being enabled, than an invalid opcode exception (#UD), instead of a floating-point exception, will be generated on the first faulting Streaming SIMD Extensions 2 instruction.

SIMD floating-point exceptions are precise and occur as soon as the instruction completes execution. They will not catch pending x87 floating-point exceptions and will not cause assertion of FERR# (independent of the value of CR0.NE). In addition, they ignore the assertion/de-assertion of IGNNE#.

2.7.1. Non-Numeric Exceptions

The Streaming SIMD Extensions 2 can generate the non-numeric exceptions listed below:

Memory Access Exceptions.

- Invalid Opcode exception (#UD).
- Stack exception (#SS).
- General protection exception (#GP). Executing Streaming SIMD Extensions 2 with an unaligned 128-bit memory reference generates a general protection exception. A 128-bit reference within the stack segment, which is not aligned to a 16-byte boundary will also generate a general protection exception, not a stack exception (#SS). However, the MOVUPD instruction, which performs a load or store, will not generate an exception for data that is not aligned to a 16-byte boundary.
- Page fault exception (#PF).
- Alignment check exception (#AC). This type of alignment check is done for operands which are less than 128-bits in size: 32-bit scalar single and 16-bit, 32-bit, and 64-bit integer MMX technology. Two, 4, or 8 byte alignments checks are possible when the Alignment check exception is enabled. Some exceptional cases are:
    - The MOVUPS instruction, which performs a 128-bit unaligned load or store; in this case, 2-, 4-, and 8-byte misalignments will be detected, but detection of 16-byte misalignment is not guaranteed and may vary with implementation.
    - The FXSAVE/FXRSTOR instructions.
    - The following three conditions that must be true to enable generation of the alignment check exception: CR0.AM flag is set, EFLAGS.AC flag is set, and CPL is 3.

System Exceptions:

- Invalid Opcode exception (#UD), when executing Streaming SIMD Extensions 2 under the following conditions (this includes both packed/scalar double-precision floating-point and 128-bit SIMD integer instructions):
    - CPUID.WNI is clear
    - CR0.EM is set (regardless of the value of CR0.TS.). Note that this does not include the PREFETCH and SFENCE instructions.
    - CR4.OSFXSR is clear. Note that this does not include the following instructions: PAVGB, PAVGW, PEXTRW, PINSRW, PMAXSW, PMAXUB, PMINSW, PMINUB, PMOVMSKB, PMULHUW, PSADBW, PSHUFW, MASKMOVQ, MOVNTQ, PREFETCH, and SFENCE.
    - Executing a Streaming SIMD Extensions instruction that causes a floating-point exception when CR4.OSXMMEXCPT = 0
- Device not available exception (#NM). Executing Streaming SIMD Extensions 2 when CR0.TS = 1 generates a DNA exception

Other exceptions can occur indirectly due to faulty execution of the above exceptions. For example, interrupt 12 occurs due to Streaming SIMD Extensions 2, and the interrupt gate directs the processor to invalid TSS (task state segment). Table 2-11 lists the causes for Interrupt 6 and Interrupt 7 with Streaming SIMD Extensions 2.

Table 2-11. Streaming SIMD Extensions 2 Exceptions

CR0.EM	CR0.TS	CR4.OSFXSR	CPUID.XMM	EXCEPTION
1	-	-	-	#UD Interrupt 6
0	1	1	1	#NM Interrupt 7
-	-	0	-	#UD Interrupt 6
-	-	-	0	#UD Interrupt 6

2.7.2. SIMD Floating-Point Exceptions

Six classes of exception conditions can occur while executing SIMD floating-point instructions:

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

Invalid, divide-by-zero and denormal exceptions are pre-computation exceptions (i.e., they are detected before any arithmetic operation occurs. Underflow, overflow and precision exceptions are post-computation exceptions.

When these floating-point exceptions occur, a processor supporting Streaming SIMD Extensions 2 takes one of two possible courses of action:

- The processor can handle the exception by itself, producing the most reasonable result and allowing numeric program execution to continue undisturbed (i.e., masked exception response).
- A software exception handler can be invoked to handle the exception (i.e., unmasked exception response).

Each of the six exception conditions described above has corresponding flag and mask bits in the MXCSR register. If an exception is masked (the corresponding mask bit in MXCSR = 1), the processor takes an appropriate default action and continues with the computation. If the exception is unmasked (mask bit = 0) and the operating system supports Streaming SIMD Extensions 2 exceptions (i.e. CR4.OSXMMEXCEPT = 1), a software exception handler is invoked immediately through Streaming SIMD Extensions 2 exception interrupt vector 19. If the exception is unmasked (mask bit = 0) and the operating system does not support Streaming

SIMD Extensions 2 exceptions (i.e. CR4.OSXMMEXCEPT = 0), an invalid opcode exception is signaled instead of a Streaming SIMD Extensions 2 exception.

Note that because Streaming SIMD Extensions 2 exceptions are precise and occur immediately, the situation does not arise where an x87 FPU instruction, an FWAIT instruction, or another Streaming SIMD Extensions and Streaming SIMD Extensions 2 will catch a pending unmasked Streaming SIMD Extensions 2 exception.

2.7.2.1. EXCEPTION PRIORITY

The processor handles exceptions according to a predetermined precedence. When a sub-operand of a packed instruction generates two or more exception conditions, the exception precedence sometimes results in the higher-priority exception being handled and the lower-priority exceptions being ignored. For example, dividing an SNaN by zero could potentially signal an invalid-arithmetic-operand exception (due to the SNaN operand) and a divide-by-zero exception. Here, if both exceptions are masked, the processor handles the higher-priority exception only (the invalid-arithmetic-operand exception), returning the quieted version of the SNaN to the destination. The prioritization policy also applies for unmasked exceptions; if both invalid and divide-by-zero are unmasked for the previous example, only the invalid flag will be set. Prioritization of exceptions is performed only on an individual sub-operand basis, and not between suboperands; for example, an invalid exception generated by one sub-operand will not prevent the reporting of a divide-by-zero exception generated by another sub-operand.

The precedence for Streaming SIMD Extensions and Streaming SIMD Extensions 2 floating-point exceptions is as follows:

1. Invalid-operation exception.
2. QNaN operand. Though this is not an exception, the handling of a QNaN operand has precedence over lower-priority exceptions. For example, a QNaN divided by zero results in a QNaN, not a zero-divide exception.
3. Any other invalid-operation exception not mentioned above or a divide-by-zero exception.
4. Denormal-operand exception. If masked, then instruction execution continues, and a lower-priority exception can occur as well.
5. Numeric overflow and underflow exceptions in conjunction with the inexact-result exception.
6. Inexact-result exception.

2.7.2.2. AUTOMATIC MASKED EXCEPTION HANDLING

If the processor detects an exception condition for a masked exception (an exception with its mask bit set), it delivers a predefined (default) response and continues executing instructions. The masked (default) responses to exceptions have been chosen to deliver a reasonable result for each exception condition and are generally satisfactory for most application code. By masking or unmasking specific floating-point exceptions in the MXCSR register, programmers can delegate responsibility for most exceptions to the processor and reserve the most severe exception conditions for software exception handlers.

Because the exception flags are "sticky," they provide a cumulative record of the exceptions that have occurred since they were last cleared. A programmer can thus mask all exceptions, run a calculation, and then inspect the exception flags to see if any exceptions were detected during the calculation.

Note that when exceptions are masked, the processor may detect multiple exceptions in a single instruction, because:

- It continues executing the instruction after performing its masked response; for example, the processor could detect a denormalized operand, perform its masked response to this exception, and then detect an underflow
- Some exceptions occur naturally in pairs, such as numeric underflow and inexact result (precision)
- Packed instructions can produce independent exceptions on each pair of operands.

Updating of exception flags is generated by a logical-OR of exception conditions for all sub-operand computations, where the OR is done independently for each type of exception; for packed computations this means 2 or 4 sub-operands (double or single precision) and for scalar computations this means 1 sub-operand (the lowest one).

2.7.2.3. HANDLING- UNMASKED EXCEPTIONS IN SOFTWARE

An application must ensure that the operating system supports unmasked exceptions before unmasking any of the exceptions in the MXCSR register (refer to Section 2.6.1.).

If the processor detects a condition for an unmasked SIMD floating-point exception, a software exception handler is invoked immediately at the end of the faulting instruction. The handler is invoked through the SIMD floating-point exception (#XF, vector 19), regardless of the state of the CR0.NE flag. If an exception is unmasked, but Streaming SIMD Extensions 2 unmasked exceptions are not enabled (CR4.OSXMMEXCPT = 0), an invalid opcode exception (#UD) is generated. However, the corresponding exception bit will still be set in the MXCSR register, as it would be if CR4.OSXMMEXCPT =1, since the invalid opcode exception handler needs to determine the cause of the exception.

A typical action of the exception handler is to store x87 FPU, Streaming SIMD Extensions, and Streaming SIMD Extensions 2 state information in memory (with the FXSAVE/FXRSTOR instructions) so that it can evaluate the exception and formulate an appropriate response. Other typical exception handler actions can include:

- Examine the stored state information to determine the nature of the error.
- Take actions to correct the condition that caused the error.
- Clear the exception bits in the x87 FPU status word or the MXCSR register.
- Return to the interrupted program and resume normal execution.

In lieu of writing recovery procedures, the exception handler can do the following:

- Increment in software an exception counter for later display or printing.
- Print or display diagnostic information (such as the XMM and MXCSR register state).
- Halt further program execution.

When an unmasked exception occurs, the processor will not alter the contents of the source register operands prior to invoking the unmasked handler. Nor will the integer EFLAGS be modified for an unmasked exception which occurs while executing the COMISD or UCOMISD instructions. Exception flags will be updated according to the following rules:

- Updating of exception flags is generated by a logical-OR of exception conditions for all sub-operand computations, where the OR is done independently for each type of exception. For packed computations this means 2 or 4 sub-operands (double or single precision), and for scalar computations this means 1 sub-operand (the lowest one).
- In the case of only masked exception conditions, all flags will be updated,
- In the case of an unmasked pre-computation type of exception condition (i.e., denormal input), all flags relating to all pre-computation conditions (masked or unmasked) will be updated, and no subsequent computation is performed (i.e., no post-computation condition can occur if there is an unmasked pre-computation condition).
- In the case of an unmasked post-computation exception condition, all flags relating to all post-computation conditions (masked or unmasked) will be updated; all pre-computation conditions, which must be masked-only will also be reported.

2.7.2.4. INTERACTION WITH X87 FLOATING-POINT EXCEPTIONS

The Streaming SIMD Extensions and Streaming SIMD Extensions 2 are independent of the x87 FPU; however, if x87 FPU applications use Streaming SIMD Extensions and Streaming SIMD Extensions 2 instructions, the following implications must be considered:

- The x87 FPU rounding mode specified in the x87 FPU control word does not affect Streaming SIMD Extensions and Streaming SIMD Extensions 2 instructions. To use the same rounding mode, the rounding control value in the MXCSR register, must be explicitly set to the same value in the x87 FPU control word.
- x87 FPU exception observability may not apply to Streaming SIMD Extensions and Streaming SIMD Extensions 2 instructions.
    - An application that expects to catch x87 FPU exceptions that occur during the execution of x87 FPU instructions will not be notified if an exception occurs in a corresponding Streaming SIMD Extensions and Streaming SIMD Extensions 2 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 the SIMD floating-point exception (#XF).
    - An application will not be able to unmask exceptions after returning from a Streaming SIMD Extensions or Streaming SIMD Extensions 2 library call to detect if an error occurred. A SIMD floating-point exception flag that was set

when the corresponding exception is unmasked will not generate a fault; only the next occurrence of that exception will generate an unmasked fault.

- An application which checks 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 within a Streaming SIMD Extensions 2 library.

2.7.3. Streaming SIMD Extensions 2 Floating-Point Exception Conditions

The following sections describe the various conditions that cause a Streaming SIMD Extensions 2 floating-point exception to be generated and the masked response of the processor when these conditions are detected.

2.7.3.1. INVALID OPERATION EXCEPTION(#I)

The invalid operation exception (#I) occurs in response to an invalid arithmetic operand. The flag (IE) for the invalid operation exception is bit 0 in the MXCSR register, and the mask bit (IM) is bit 7 of the MXCSR register.

If the invalid operation exception is masked, the processor returns the double-precision indefinite value to the destination operand. This value overwrites the destination register specified by the instruction.

If the invalid operation exception is not masked, a software exception handler is invoked (see Section 2.7.2.3.) and the operands remain unchanged.

The processor can detect a variety of invalid arithmetic operations that can be coded in a program. These operations generally indicate a programming error, such as dividing ∞ by ∞. Table 2-12 lists the invalid arithmetic operations that the processor detects for Streaming SIMD Extensions 2 instructions. This group includes the invalid operations defined in IEEE Standard 754.

The invalid operation exception is not affected by the flush to zero mode.

Table 2-12. Invalid Arithmetic Operations and The Masked Responses to Them

Condition	Masked Response
ADDPD/ADDSD/DIVPD/DIVSD/ MULPD/MULSD/SUBPD/SUBSD with a SNaN.	Return the signaling NaN converted to a quiet NaN; Refer to Table 2-3 for more details; set #I flag.
CMPPD/CMPSD with QNaN/SNaN operands	Return a mask of all 0s (except for the predicates “ne” and “unordered”, which returns a mask of all 1s); set #I flag.
COMISD with QNaN/SNaN operand(s).	Set EFLAGS values to “not comparable”; set #I flag.
UCOMISD with QNaN operand(s).	Set EFLAGS values to “not comparable”.
MAXPD/MAXSD/MINPD/MINSD with QNaN/SNaN operand(s).	Return the src2 value; set #I flag.
SQRTPD/SQRTSD with SNaN operand(s).	Return the SNaN converted to a QNaN; set #I flag;
UCOMISD with QNaN operand(s).	Set EFLAGS values to “not comparable”.
UCOMISD with SNaN operand(s).	Set EFLAGS values to ”not comparable”; set #I flag.
Addition of opposite signed infinities or subtraction of like-signed infinities.	Return the QNaN Indefinite; set #I flag.
Multiplication of infinity by zero.	Return the QNaN Indefinite; set #I flag.
Divide of (0/0) or(∞/∞)	Return the QNaN Indefinite; set #IA flag.
SQRTPD of negative operands (except zero).	Return the QNaN Indefinite; set #IA flag.
Conversion to integer when the source register is a NaN, Infinity or exceeds the representable range.	Return the Integer Indefinite; set #IA flag.

2.7.3.2. DIVISION-BY-ZERO EXCEPTION (#Z)

The processor reports a divide-by-zero exception whenever a DIVPD or DIVSD instruction attempts to divide a finite non-zero operand by 0. The flag (ZE) for the divide-by-zero exception is bit 2 of the MXCSR register, and the mask bit (ZM) is bit 9 of MXCSR.

The masked response for divide-by-zero exception is to set the ZE flag in the MXCSR register and return an infinity signed with the exclusive OR of the sign of the operands. If the divide-by-zero exception is not masked, the ZE flag is set, a software exception handler is invoked (see Section 2.7.2.3.) and the source operands remain unchanged.

The divide by zero exception is not affected by the flush to zero mode.

2.7.3.3. 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) for the denormal-operand exception is bit 1 of the MXCSR register, and the mask bit (DM) is bit 8 of MXCSR

When a denormal-operand exception occurs and the exception is masked, the processor sets the DE flag, then proceeds with the instruction. Operating on denormal numbers will produce results at least as good as, and often better than, what can be obtained when denormal numbers are flushed to zero. Programmers can mask this exception so that a computation may proceed, then analyze any loss of accuracy when the final result is delivered.

When a denormal-operand exception occurs and the exception is not masked, the processor sets the DE bit in the MXCSR register and a software exception handler is invoked (see Section 2.7.2.3.). The source operands remain unchanged. When denormal operands have reduced significance due to loss of low-order bits, it may be advisable to not operate on them. Precluding denormal operands from computations can be accomplished by an exception handler that responds to unmasked denormal-operand exceptions.

Conversion instructions (CVTPI2PD, CVTPD2PI, CVTTPD2PI, CVTDQ2PD, CVTPD2DQ, CVTTPD2DQ, CVTSI2SD, CVTSD2SI, CVTTSD2SI) do not signal denormal exceptions.

The denormal operand exception is not affected by the flush to zero mode.

2.7.3.4. NUMERIC OVERFLOW EXCEPTION (#O)

The processor reports a floating-point numeric overflow exception whenever the rounded result of an arithmetic instruction exceeds the largest allowable finite value that will fit into the destination operand. This is possible with ADDPD, ADDSD, SUBPD, SUBSD, MULPD, MULSD, DIVPD, DIVSD, CVTPD2PS, CVTSD2SS. The flag (OE) for the numeric overflow exception is bit 3 of the MXCSR register, and the mask bit (OM) is bit 10 of MXCSR.

When a numeric overflow exception occurs and the exception is masked, the processor sets the OE and PE flags in the MXCSR register and returns one of the values shown in Table 2-13 according to the current rounding mode being used (see Section 2.5.5.).

When a numeric overflow exception occurs and the exception is unmasked, the operands are left unaltered and a software exception handler is invoked (see Section 2.7.2.3.). The OE flag in the MXCSR register is set; the PE flag is only set if a loss of accuracy has occurred in addition to the overflow.

The numeric overflow exception is not affected by the flush to zero mode.

Table 2-13. Masked Responses to Numeric Overflow

Rounding Mode	Sign of True Result	Result
To nearest	+	+∞
	–	–∞
Toward –∞	+	Largest finite positive number
	–	–∞
Toward +∞	+	+∞
	–	Largest finite negative number
Toward zero	+	Largest finite positive number
	–	Largest finite negative number

2.7.3.5. NUMERIC UNDERFLOW EXCEPTION (#U)

The processor reports a floating-point numeric underflow exception whenever the rounded result of an arithmetic instruction is tiny; that is, less than the smallest possible normalized, finite value that will fit into the destination operand. The Underflow exception can occur in the execution of the instructions ADDPD, ADDSD, SUBPD, SUBSD, MULPD, MULSD, DIVPD, DIVSD, CVTPD2PS, CVTSD2SS. The flag (UE) for the numeric underflow exception is bit 4 of the MXCSR register and the mask bit (UM) is bit 11 of MXCSR.

Two related events contribute to underflow:

- Creation of a tiny result which, because it is so small, may cause some other exception later (such as overflow upon division).
- Creation of an inexact result; i.e. the delivered result differs from what would have been computed were both the exponent and precision unbounded.

Which of these events triggers the underflow exception depends on whether the underflow exception is masked:

- Underflow exception masked. The underflow exception is signaled when the result is both tiny and inexact.
- Underflow exception not masked: The underflow exception is signaled when the result is tiny, regardless of inexactness.

The response to an underflow exception also depends on whether the exception is masked:

- Masked response. The result is normal, denormal or zero. The precision exception is also triggered.
- Unmasked response. If the original computation generated an imprecise mantissa, the inexact (PE) status flag will be set. In either case, the operands are left unaltered and a software exception handler is invoked (see Section 2.7.2.3.).

If underflow is masked and flush to zero mode is enabled, an underflow condition will set the underflow (UE) and inexact (PE) status flags and return a correctly signed zero result; this will avoid the performance penalty associated with generating a denormalized result. If underflow is unmasked, the flush to zero mode is ignored and an underflow condition be handled as described above.

2.7.3.6. 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 inexact-result exception flag (PE) is bit 5 of the MXCSR register, and the mask bit (PM) is bit 12 of MXCSR.

If the inexact-result exception is masked when an inexact-result condition occurs and a numeric overflow or underflow condition has not occurred, the processor sets the PE flag and stores the rounded result in the destination operand. The current rounding mode determines the method used to round the result (refer to Section 2.5.5.).

If the inexact-result exception is not masked when an inexact result occurs and numeric overflow or underflow has not occurred, the operands are left unaltered and a software exception handler is invoked (see Section 2.7.2.3.).

If an inexact result occurs in conjunction with numeric overflow or underflow, one of the following operations is carried out:

- If an inexact result occurs along with masked overflow or underflow, the OE or UE flag and the PE flag are set and the result is stored as described for the overflow or underflow exceptions (see Section 2.7.3.4.. or Section 2.7.3.5.). If the inexact result exception is unmasked, the processor also invokes the software exception handler.
- If an inexact result occurs along with unmasked overflow or underflow and the destination operand is a register, the OE or UE flag and the PE flag are set, the result is stored as described for the overflow or underflow exceptions, and the software exception handler is invoked.

In flush to zero mode, the inexact result exception is reported.

2.8. SYSTEM PROGRAMMING MODEL

The following sections describe the interface of the Streaming SIMD Extensions 2 to the operating system.

2.8.1. Enabling Streaming SIMD Extensions 2 Support

Certain steps must be taken in both applications and the operating system to check if the processor supports Streaming SIMD Extensions 2 and associated unmasked exceptions. This section describes this process, which is conducted using the bits described in Table 2-14 and Table 2-15.

Table 2-14. CPUID bits for Streaming SIMD Extensions 2 Support

CPUID bit (EAX = 1)	Meaning
FXSR (EDX bit24)	If set, CPU supports FXSAVE/FXRSTOR. The operating system can read this bit to determine if it can use FXSAVE/FXRSTOR in place of FSAVE/FRSTOR for context switches.
WNI (EDX bit26)	If set, the Streaming SIMD Extensions 2 are supported by the processor.

Table 2-15. CR4 bits for Streaming SIMD Extensions 2 Support

CR4 bit	Meaning
OSFXSR (bit9)	Defaults to clear. If both the CPU and the operating system support FXSAVE/FXRSTOR for use during context switches, then the operating system will set this bit.
OSXMMEXCPT (bit10)	Defaults to clear. The operating system will set this bit if it supports both unmasked Streaming SIMD Extensions and Streaming SIMD Extensions 2 exceptions.

If the operating system is going to use the FXSAVE/FXRSTOR instructions, it should first check CPUID.FXSR to determine if the processor supports these instructions. If the processor does support FXSAVE/FXRSTOR, then the operating system can set CR4.OSFXSR without faulting and enable code for context switching that uses FXSAVE/FXRSTOR instead of FSAVE/FRSTOR.

At this point, if the operating system also supports unmasked SIMD floating-point exceptions, it should check CPUID.WNI to see if this is a Streaming SIMD Extensions 2 enabled processor. If CPUID.WNI is set, this verifies that the operating system can set CR4.OSXMMEXCPT without faulting. When a Streaming SIMD Extensions 2 floating-point exception occurs, the processor will generate a Streaming SIMD floating-point exception (#XF) if CR4.OSXMMEXCPT=1, otherwise the processor will generate an invalid operation exception (#UD).

The process by which an application detects the existence of Streaming SIMD Extensions 2 is described in Section 2.6.1., “Using the CPUID Instruction to Detecting the Existence of the Streaming SIMD Extensions 2”.

2.8.2. Device Not Available Exception

Streaming SIMD Extensions 2 will cause a device not available exception (#NM) if the processor attempts to executes Streaming SIMD Extensions 2 while CR0.TS is set. If CPUID.WNI is clear, execution of any Streaming SIMD Extensions 2 instructions will cause an invalid opcode exception (#UD) regardless of the state of CR0.TS.

2.8.3. Streaming SIMD Extensions 2 Emulation

The CR0.EM bit (used when emulating floating-point instructions) cannot be used in the same way for MMX technology emulation. If Streaming SIMD Extensions 2 executes when the CR0.EM bit is set, an invalid opcode exception (#UD) is generated instead of a device not available exception (#NM).

2.8.4. Numeric Error flag and IGNNE#

Streaming SIMD Extensions 2 ignore CR0.NE (treats it as if it were always set) and the IGNNE# pin and always uses the SIMD floating-point exception (#XF) for error reporting.

CHAPTER 3
STREAMING SIMD EXTENSIONS 2 INSTRUCTION SET

The new instructions added to the IA-32 architecture with the Streaming SIMD Extensions 2 are divided into three sections:

- Packed and scalar double-precision floating-point instructions (see Section 3.2., "Packed and Scalar Double-Precision Floating-Point Instructions").
- SIMD integer instructions that operate on 128-bit operands (see Section 3.3., "SIMD Integer Instructions").
- Cacheability control and memory ordering instructions (see Section 3.4., "Cacheability Control and Memory Ordering Instructions").

The Streaming SIMD Extensions 2 also modified the operation of several existing IA-32 instructions (see Section 3.5., "Modified Instructions").

Appendix A summarizes all the Streaming SIMD Extensions 2 instructions, grouped into the three categories describe above.

3.1. NOTATION

The following notation is used in describing the Streaming SIMD Extensions 2 assembly-code instruction operands:

- **r32:** General-purpose register.
- **xmm/m128:** indicates an XMM register or a 128-bit memory location.
- **xmm/m64:** indicates an XMM register or a 64-bit memory location.
- **xmm/m32:** indicates an XMM register or a 32-bit memory location.
- **mm/m64:** indicates an MMX™ register or a 64-bit memory location.
- **xmm/m128:** indicates an MMX register or a 128-bit memory location.
- **imm8:** indicates an immediate 8-bit operand.
- **ib:** indicates that an immediate byte operand follows the opcode, ModR/M byte or scaled-indexing byte.

When there is ambiguity, xmm1 indicates the first source operand and xmm2 the second source operand.

Table 3-1 describes the naming conventions used in the Streaming SIMD Extensions 2 instructions mnemonics.

Table 3-1. Key to Streaming SIMD Extensions 2 instructions Technology Naming Convention

Mnemonic	Description
DQ	Packed integer double quadword (for example, xmm0)
PD	Packed double-precision floating-point (for example, xmm0)
SI	Scalar integer (for example, EAX)
SD	Scalar double-precision floating-point (for example, low 64 bits of xmm0)

3.2. PACKED AND SCALAR DOUBLE-PRECISION FLOATING-POINT INSTRUCTIONS

The following instructions were added to the IA-32 architecture to support packed and scalar double-precision floating-point data in the XMM registers.

ADDPD—Packed Double-Precision Floating-Point Add

Instruction	Description
ADDPD xmm1, xmm2/m128	Add packed double-precision floating-point numbers from xmm2/m128 to xmm1.

Description

Performs a packed addition of the two packed double-precision floating-point numbers.

Operation

```
xmm1[63-0]   = xmm1[63-0]   + xmm2/m128[63-0];
xmm1[127-64] = xmm1[127-64] + xmm2/m128[127-64];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments
	If memory operand is not aligned on a 16-byte boundary, regardless of segment
#SS(0)	For an illegal address in the SS segment;
#PF(fault-code)	For a page fault;
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1);
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0)
	If CR0.EM = 1
	If CR4.OSFXSR(bit 9) = 0;
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH
#NM	If TS bit in CR0 is set

#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1)
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0)
	If CR0.EM = 1
	If CR4.OSFXSR (bit 9) = 0
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

ADDSD—Scalar Double-Precision Floating-Point Add

Instruction	Description
ADDSD xmm1, xmm2/m64	Add the low double-precision floating-point number from xmm2/mem64 to xmm1.

Description

Adds the low double-precision floating-point numbers of both operands; the high quadword element of the xmm1operand is passed through to the result.

Operation

```
xmm1[63-0]   = xmm1[63-0] + xmm2/m64[63-0];
xmm1[127-64] = xmm1[127-64];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).

#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault;
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

ANDNPD—Bitwise Logical AND NOT for Double-Precision Floating-Point

Instruction	Description
ANDNPD xmm1, xmm2/m128	Inverts the 128 bits of xmm1, then performs bitwise AND of result with the 128 bits of xmm2/mem128.

Description

Returns a bitwise logical AND of xmm2/mem128 and the complement of xmm1.

Operation

```
xmm1[127-0] = ~(xmm1[127-0]) & xmm2/m128[127-0];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.

#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

ANDPD—Bitwise Logical AND for Double-Precision Floating-Point

Instruction	Description
ANDPD xmm1, xmm2/m128	Bitwise logical AND of xmm2/mem128 and xmm1.

Description

Returns a bitwise logical AND between xmm1 and xmm2/mem128.

Operation

```
xmm1[127-0] &= xmm2/m128[127-0];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).

#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

CMPPD—Packed Double-Precision Floating-Point Compare

Instruction	Description
CMPPD xmm1, xmm2/m128, imm8	Compare packed double-precision floating-point numbers from xmm2/mem128 with packed double-precision floating-point numbers in xmm1, using imm8 as predicate.

Description

Compares the individual pairs of double-precision floating-point numbers in the source operands and returns an all 1s 64-bit mask or an all 0s 64-bit mask according to the comparison predicate specified by imm8. Note that a subsequent computational instruction that uses this mask as an input operand will not generate a fault, since a mask of all 0s corresponds to a floating-point number of +0.0 and a mask of all 1s corresponds to a floating-point number of -QNaN. The following table shows the different comparison types:

Predicate	Description	Relation	Emulation	imm8 Encoding	Result if NaN Operand	QNaN Operand Signals Invalid
eq	equal	xmm1 == xmm2		000B	False	No
lt	less-than	xmm1 < xmm2		001B	False	Yes
le	less-than-or-equal	xmm1 <= xmm2		010B	False	Yes
	greater than	xmm1 > xmm2	swap, protect, lt		False	Yes
	greater-than-or-equal	xmm1 >= xmm2	swap protect, le		False	Yes
unord	unordered	xmm1 ? xmm2		011B	True	No
neq	not-equal	!(xmm1 == xmm2)		100B	True	No
nlt	not-less-than	!(xmm1 < xmm2)		101B	True	Yes
nle	not-less-than-or-equal	!(xmm1 <= xmm2)		110B	True	Yes
	not-greater-than	!(xmm1 > xmm2)	swap, protect, nlt		True	Yes
	not-greater-than-or-equal	!(xmm1 >= xmm2)	swap, protect, nle		True	Yes
ord	ordered	!(xmm1 ? xmm2)		111B	False	No

Some of the comparisons can be achieved only through software emulation. For these comparisons the programmer must swap the operands, copying registers when necessary to protect the data that will now be in the destination, and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in under the heading “Emulation.”

Note that the greater-than, greater-than-or-equal, not-greater-than, and not-greater-than-or-equal relations are not directly implemented in hardware.

Operation

```
switch (imm8) {
    case eq:          op = eq;
    case lt:          op = lt;
    case le:          op = le;
    case unord:       op = unord;
    case neq:         op = neq;
    case nlt:         op = nlt;
    case nle:         op = nle;
    case ord:         op = ord;
    default:          Reserved;
    }

cmp0 = op(xmm1[63-0],xmm2/m128[63-0]);
cmp1 = op(xmm1[127-64],xmm2/m128[127-64]);

xmm1[63-0]   = (cmp0) ? 0xffffffffffffffff : 0x0000000000000000;
xmm1[127-64] = (cmp1) ? 0xffffffffffffffff : 0x0000000000000000;
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

Invalid if SNaN operand, invalid if QNaN and predicate as listed in above table, denormal.

Comments

Compilers and assemblers should implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction.

Pseudo-Op	Implementation
CMPEQPD xmm1, xmm2	CMPPD xmm1,xmm2, 0
CMPLTPD xmm1, xmm2	CMPPD xmm1,xmm2, 1
CMPLEPD xmm1, xmm2	CMPPD xmm1,xmm2, 2
CMPUNORDPD xmm1, xmm2	CMPPD xmm1,xmm2, 3
CMPNEQPD xmm1, xmm2	CMPPD xmm1,xmm2, 4
CMPNLTPD xmm1, xmm2	CMPPD xmm1,xmm2, 5
CMPNLEPD xmm1, xmm2	CMPPD xmm1,xmm2, 6
CMPORDPD xmm1, xmm2	CMPPD xmm1,xmm2, 7

The greater-than relations not implemented in hardware require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Bits 7-4 of the immediate field are reserved. Different processors may handle them differently. Usage of these bits risks incompatibility with future processors.

CMPSD—Scalar Double-Precision Floating-Point Compare

Instruction	Description
CMPSD xmm1, xmm2/m64, imm8	Compare low double-precision floating-point number from xmm2/mem64 with low double-precision floating-point number in xmm1 register using imm8 as predicate.

Description

Compares the low double-precision floating-point numbers in the source operands and returns an all 1s 64-bit mask or an all 0s 64-bit mask according to the comparison predicate specified by imm8. The values for the high double-precision floating-point numbers in the source operands are not compared. Note that a subsequent computational instruction that uses this mask as an input operand will not generate a fault, since a mask of all 0s corresponds to a floating-point number of +0.0 and a mask of all 1s corresponds to a floating-point number of -QNaN. The following table shows the different comparison types:

Predicate	Description	Relation	Emulation	imm8 Encoding	Result if NaN Operand	QNaN Operand Signals Invalid
eq	equal	xmm1 == xmm2		000B	False	No
lt	less-than	xmm1 < xmm2		001B	False	Yes
le	less-than-or-equal	xmm1 <= xmm2		010B	False	Yes
	greater than	xmm1 > xmm2	swap, protect, lt		False	Yes
	greater-than-or-equal	xmm1 >= xmm2	swap protect, le		False	Yes
unord	unordered	xmm1 ? xmm2		011B	True	No
neq	not-equal	!(xmm1 == xmm2)		100B	True	No
nlt	not-less-than	!(xmm1 < xmm2)		101B	True	Yes
nle	not-less-than-or-equal	!(xmm1 <= xmm2)		110B	True	Yes
	not-greater-than	!(xmm1 > xmm2)	swap, protect, nlt		True	Yes
	not-greater-than-or-equal	!(xmm1 >= xmm2)	swap, protect, nle		True	Yes
ord	ordered	!(xmm1 ? xmm2)		111B	False	No

Some of the comparisons can be achieved only through software emulation. For these comparisons the programmer must swap the operands, copying registers when necessary to protect the data that will now be in the destination, and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in under the heading “Emulation”.

Note that the greater-than, greater-than-or-equal, not-greater-than, and not-greater-than-or-equal relations are not directly implemented in hardware.

Operation

```
<algorithm1>
<algorightm_next>
switch (imm8) {
     case eq:          op = eq;
     case lt:          op = lt;
     case le:          op = le;
     case unord:       op = unord;
     case neq:         op = neq;
     case nlt:         op = nlt;
     case nle:         op = nle;
     case ord:         op = ord;
     default:          Reserved;
     }
cmp0 = op(xmm1[63-0],xmm2/m64[63-0]);

xmm1[63-0]   = (cmp0) ? 0xffffffffffffffff : 0x0000000000000000;
xmm1[127-64] = xmm1[127-64];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Invalid if SNaN operand, invalid if QNaN and predicate as listed in above table, denormal.

Comments

Compilers and assemblers should implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction.

Pseudo-Op	Implementation
CMPEQSD xmm1, xmm2	CMPSD xmm1,xmm2, 0
CMPLTSD xmm1, xmm2	CMPSD xmm1,xmm2, 1
CMPLESD xmm1, xmm2	CMPSD xmm1,xmm2, 2
CMPUNORDSD xmm1, xmm2	CMPSD xmm1,xmm2, 3
CMPNEQSD xmm1, xmm2	CMPSD xmm1,xmm2, 4
CMPNLTSD xmm1, xmm2	CMPSD xmm1,xmm2, 5
CMPNLESD xmm1, xmm2	CMPSD xmm1,xmm2, 6
CMPORDSD xmm1, xmm2	CMPSD xmm1,xmm2, 7

The greater-than relations not implemented in hardware require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

COMISD—Scalar Ordered Double-Precision Floating-Point Compare and Set EFLAGS

Instruction	Description
COMISD xmm1, xmm2/m64	Compare low double-precision floating-point number in xmm1 register with low double-precision floating-point number in xmm2/mem64 and set the EFLAGS flags register accordingly.

Description

Compares two double-precision floating-point numbers and sets the ZF, PF, and CF flags in the EFLAGS register as described in "Operation." Although the data type is packed double-precision floating-point, only the low values in each source operand are compared. In addition, the OF, SF and AF flags in the EFLAGS register are zeroed out. The unordered predicate is returned if either source operand is a NaN (QNaN or SNaN).

Operation

```
switch (xmm1[63-0] <> xmm2/m64[63-0]) {
    case UNORDERED:     ZF,PF,CF = 111;
    case GREATER_THAN:  ZF,PF,CF = 000;
    case LESS_THAN:     ZF,PF,CF = 001;
    case EQUAL:         ZF,PF,CF = 100;
    OF,AF,SF = 0;
    }
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Invalid (if SNaN or QNaN operands), denormal. The EFLAGS register is not updated in the presence of unmasked numeric exceptions.

Comments

The COMISD instruction differs from the UCOMISD instruction in that it signals an invalid numeric exception when a source operand is either a QNaN or SNaN. The UCOMISD instruction signals invalid only if a source operand is an SNaN.

The use of Repeat (F2H, F3H) prefixes with COMISD is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with COMISD risks incompatibility with future processors.

CVTDQ2PD—Packed Doubleword Signed Integer to Packed Double-Precision Floating-Point Conversion

Instruction	Description
CVTDQ2PD xmm1, xmm2/m64	Convert two packed doubleword signed integers from xmm2/m128 to two packed double-precision floating-point numbers.

Description

Converts two packed doubleword signed integers to two packed double-precision floating-point numbers. When converting from a memory operand, only 64 bits are used by the instruction.

Operation

```
xmm1[63-0]   = (double) (xmm2/m64[31-0]);
xmm1[127-64] = (double) (xmm2/m64[63-32]);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).

#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

None.

CVTPD2PI—Packed Double-Precision Floating-Point to Packed Doubleword Integer Conversion

Instruction	Description
CVTPD2PI mm, xmm/m128	Convert two packer double-precision floating-point numbers from xmm/m128 to two packed doubleword signed integers in mm using rounding specified by MXCSR.

Description

Converts two packed double-precision floating-point numbers in xmm/m128 to two signed packed doubleword integers in mm. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Operation

```
mm[31-0]   = (int) (xmm/m128[63-0]);
mm[63-32]  = (int) (xmm/m128[127-64]);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#MF	If there is a pending FPU exception.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#MF	If there is a pending FPU exception.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

Invalid, Precision.

Comments

This instruction behaves identically to original MMX instructions, in the presence of x87 FPU instructions, including:

- Transition from x87 FPU to MMX technology operations (TOS=0, floating point valid bits set to all valid).
- MMX technology instructions write ones (1s) to the exponent part of the corresponding x87 FPU register.

Prioritization for fault and assist behavior for the CVTPS2PI instruction is as follows:

- Memory source
  a. Invalid opcode (CR0.EM=1)
  b. DNA (CR0.TS=1)
  c. #MF, pending x87 FPU fault signaled
  d. After returning from #MF, x87 FPU or MMX technology operation transition
  e. #SS or #GP, for limit violation
  f. #PF, page fault

g. 8) Streaming SIMD Extensions numeric fault (i.e., invalid, precision)

- Register source

a. Invalid opcode (CR0.EM=1)

b. DNA (CR0.TS=1)

c. #MF, pending x87 FPU fault signaled

d. After returning from #MF, x87 FPU or MMX technology operation transition

e. Streaming SIMD Extensions numeric fault (i.e., precision)

CVTPD2DQ—Packed Double-Precision Floating-Point to Packed Doubleword Integer Conversion

Instruction	Description
CVTPD2PQ xmm1, xmm2/m128	Convert two packed double-precision floating-point numbers from xmm2/m128 to two packed doubleword signed integers in xmm1 using rounding specified by MXCSR.

Description

Converts two packed double-precision floating-point numbers in xmm/m128 to two packed signed doubleword integers in the two low doublewords of xmm1; the high quadword of xmm1 is set to all 0s. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Operation

```
xmm1[31-0]   = (int) (xmm2/m128[63-0]);
xmm1[63-32]  = (int) (xmm2/m128[127-64]);
xmm1[95-64]  = 0x00000000;
xmm1[127-96] = 0x00000000;
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

Invalid, Precision.

CVTPD2PS—Packed Double-Precision Floating-Point to Packed Single-Precision Floating-Point Conversion

Instruction	Description
CVTPD2PS xmm1, xmm2/m128	Convert two double-precision floating-point numbers to two single-precision floating-point numbers.

Description

Converts two double-precision floating-point numbers to two single-precision floating-point numbers, and stores the result in the two low doublewords of xmm1. The bits in the high quad-word of xmm1 are cleared.

Operation

```
xmm1[31-0]   = (float) (xmm2/m128[63-0]);
xmm1[63-32]  = (float) (xmm2/m128[127-64]);
xmm1[95-64]  = 0x00000000;
xmm1[127-96] = 0x00000000;
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

CVTPI2PD—Packed Signed Doubleword Integer to Packed Double-Precision Floating-Point Conversion

Instruction	Description
CVTPI2PD xmm, mm/m64	Convert lowest two signed doubleword integers from mm/mem64 to two double-precision floating-point numbers.

Description

Converts two signed doubleword integers to two packed double-precision floating-point numbers. When converting from a memory operand, only 64 bits are used by the instruction.

Operation

```
xmm[63-0]   = (double) (mm/m64[31-0]);
xmm[127-64] = (double) (mm/m64[63-32]);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#MF	If there is a pending FPU exception.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#MF	If there is a pending FPU exception.

#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

Comments

This instruction behaves identically to an MMX instructions in the presence of an x87 FPU instructions, including:

- Transition from x87 FPU to MMX technology operations (TOS=0, floating point valid bits set to all valid).
- MMX instructions write ones (1's) to the exponent part of the corresponding x87 FPU register.

However, the use of a memory source operand with this instruction will not result in the above transition from x87 FPU to MMX technology.

Prioritization for fault and assist behavior for the CVTPI2PS instruction is as follows:

- Memory source
  - a. 1) Invalid opcode (CR0.EM=1)
  - b. 2) DNA (CR0.TS=1)
  - c. 3) #SS or #GP, for limit violation
  - d. 4) #PF, page fault
  - e. 5) Streaming SIMD Extensions numeric fault (i.e., precision)

- Register source

  a. 1) Invalid opcode (CR0.EM=1)

  b. 2) DNA (CR0.TS=1)

  c. 3) #MF, pending x87 FPU fault signaled

  d. 4) After returning from #MF, x87 FPU or MMX technology transition

  e. 5) Streaming SIMD Extensions numeric fault (i.e., precision)

CVTPS2PD—Packed Single-Precision to Packed Double-Precision Floating-Point Conversion

Instruction	Description
CVTPS2PD xmm1, xmm2/m64	Convert two single-precision floating-point numbers to two double-precision floating-point numbers.

Description

Converts the two low single-precision floating-point numbers from xmm2/m64 to two packed double-precision floating-point numbers, and returns the result to xmm1.

Operation

```
xmm1[63-0]   = (double) (xmm2/m64[31-0]);
xmm1[127-64] = (double) (xmm2/m64[63-32]);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).

#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Invalid, Denormal.

CVTSD2SI—Scalar Double-Precision Floating-Point to Signed Doubleword Integer Conversion

Instruction	Description
CVTSD2SI r32, xmm/m64	Convert one double-precision floating-point number from xmm/m64 to one doubleword signed integer using the rounding mode specified by MXCSR, and return the result to a general-purpose register.

Description

Converts the low double-precision floating-point number in xmm or a double-precision floating-point number from memory to a signed doubleword integer, and returns the result to a general-purpose register. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Operation

```
r32 = (int) (xmm/m64[63-0]);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.

#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Invalid, Precision.

CVTSD2SS—Scalar Double-Precision Floating-Point to Scalar Single-Precision Floating-Point Conversion

Instruction	Description
CVTSD2SS xmm1, xmm2/m64	Convert double-precision floating-point number in xmm2/m64 to single-precision floating-point number in xmm1.

Description

Converts the low double-precision floating-point number in xmm2 or a to a double-precision floating-point number in memory to a single-precision floating-point number, and returns the result to the low doubleword of xmm1. The upper 3 doublewords in the destination register are left unchanged.

Operation

```
xmm1[31-0]   = (float) (xmm2/m64[63-0]);
xmm1[63-32]  = xmm1[63-32];
xmm1[95-64]  = xmm1[95-64];
xmm1[127-96] = xmm1[127-96];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.

#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

CVTSI2SD—Scalar signed INT32 to Double-Precision Floating-Point Conversion

Instruction	Description
CVTSI2SD xmm, r/m32	Convert signed doubleword integer from r/m32 to a double-precision floating-point number.

Description

Converts a signed doubleword integer from a general-purpose register or from memory to a double-precision floating-point number, and stores the result in xmm. The high quadword of the destination register is left unchanged.

Operation

```
xmm[63-0]   = (double) (r/m32);
xmm[127-64] = xmm[127-64];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.

#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

None.

CVTSS2SD—Scalar Single-Precision Floating-Point to Scalar Double-Precision Floating-Point Conversion

Instruction	Description
CVTSS2SD xmm1, xmm2/m32	Convert single-precision floating-point number to double-precision floating-point number.

Description

Converts the single-precision floating-point number from the low doubleword of xmm2 or memory to a double-precision floating-point number, and returns it to the low quadword of xmm1. The high quadword of the destination register is left unchanged.

Operation

```
xmm1[63-0]   = (double) (xmm2/m32[31-0]);
xmm1[127-64] = xmm1[127-64];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.

#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Invalid, Denormal.

CVTTPD2PI—Packed Double-Precision Floating-Point to Packed INT32 Conversion (truncate)

Instruction	Description
CVTTPD2PI mm, xmm/m128	Convert two double-precision floating-point numbers from xmm2/m128 to two signed doubleword integers in mm using truncate.

Description

Converts the two double-precision floating-point numbers in xmm/m128 to two doubleword signed integers in mm. If the conversion is inexact, the truncated result is returned.

Operation

```
mm[31-0]   = (int) (xmm/m128[63-0]);
mm[63-32]  = (int) (xmm/m128[127-64]);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#MF	If there is a pending FPU exception.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.

#MF	If there is a pending FPU exception.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

Invalid, Precision.

Comments

This instruction behaves identically to an MMX instruction, in the presence of x87 FPU instructions, including:

- Transition from x87 FPU to MMX technology operations (TOS=0, floating point valid bits set to all valid).
- MMX instructions write ones (1's) to the exponent part of the corresponding x87 FPU register.

Prioritization for fault and assist behavior for the CVTPI2PS instruction is as follows:

- Memory source
  a. Invalid opcode (CR0.EM=1)
  b. DNA (CR0.TS=1)
  c. #MF, pending x87 FPU fault signaled
  d. After returning from #MF, x87 FPU or MMX technology operation transition
  e. #SS or #GP, for limit violation
  f. #PF, page fault
  g. Streaming SIMD Extensions numeric fault (i.e., invalid, precision)
- Register source

a. Invalid opcode (CR0.EM=1)

b. DNA (CR0.TS=1)

c. #MF, pending x87 FPU fault signaled

d. After returning from #MF, x87 FPU or MMX technology operation transition

e. Streaming SIMD Extensions numeric fault (i.e., precision)

CVTTPD2DQ—Packed Double-Precision Floating-Point to Packed Doubleword Integer Conversion (Truncate)

Instruction	Description
CVTTPD2DQ xmm1, xmm2/m128	Convert two double-precision floating-point numbers from xmm2/m128 to two signed doubleword integers in XMM using truncate.

Description

Converts the two double-precision floating-point numbers in xmm/m128 to two doubleword signed integers, and returns the result to the two low doublewords of xmm1; the high 64 bits of xmm1 are set to all 0s. If the conversion is inexact, the truncated result is returned.

Operation

```
xmm1[31-0]   = (int) (xmm2/m128[63-0]);
xmm1[63-32]  = (int) (xmm2/m128[127-64]);
xmm1[95-64]  = 0x00000000;
xmm1[127-96] = 0x00000000;
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

Invalid, Precision.

CVTTSD2SI—Scalar Double-Precision Floating-Point to Signed Doubleword Integer Conversion (Truncate)

Instruction	Description
CVTTSD2SI r32, xmm/m64	Convert one double-precision floating-point number from xmm/m64 to one signed doubleword integer using truncate, and return the result to r32.

Description

Converts the double-precision floating-point number in the low quadword of xmm or from memory to a signed doubleword integer, and returns the result to a general-purpose register. If the conversion is inexact, the truncated result is returned.

Operation

```
r32 = (int) (xmm/m64[63-0]);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.

#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Invalid, Precision.

DIVPD—Packed Double-Precision Floating-Point Divide

Instruction	Description
DIVPD xmm1, xmm2/m128	Divide packed double-precision floating-point numbers in xmm1 by xmm2/mem128

Description

Divides the packed double-precision floating-point numbers in xmm2/m128 by the packed double-precision floating-point numbers in xmm1.

Operation

```
xmm1[63-0]   = xmm1[63-0]   / (xmm2/m128[63-0]);
xmm1[127-64] = xmm1[127-64] / (xmm2/m128[127-64]);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.

#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

Overflow, Underflow, Divide-By-Zero, Invalid, Precision, Denormal.

DIVSD—Scalar Double-Precision Floating-Point Divide

Instruction	Description
DIVSD xmm1, xmm2/m64	Divide low double-precision floating-point numbers in xmm1 by xmm2/mem64

Description

Divides the double-precision floating-point number in the low quadword of xmm2 or from memory by the double-precision floating-point number in the low quadword of xmm1, and returns the result to the low quadword of xmm1. The high quadword of the destination operand is left unchanged.

Operation

```
xmm1[63-0]   = xmm1[63-0] / (xmm2/m64[63-0]);
xmm1[127-64] = xmm1[127-64];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	If CR0.EM = 1.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Overflow, Underflow, Divide-By-Zero, Invalid, Precision, Denormal.

MAXPD—Packed Double-Precision Floating-Point Maximum

Instruction	Description
MAXPD xmm1, xmm2/m128	Return the maximum double-precision floating-point numbers between xmm2/mem128 and xmm1.

Description

Returns the maximum double-precision floating-point numbers from the pairs of elements in xmm1 and xmm2/mem128. If the values being compared are both zeros, source operand 2 (xmm2/m128) is returned. If a value in source operand 2 is an SNaN, that SNaN is forwarded unchanged to the destination (i.e., a quieted version of the SNaN is not returned).

Operation

```
xmm1[63-0]   =
              (xmm1[63-0] == NAN) ? xmm2[63-0] :
              xmm2[63-0] == NAN) ? xmm2[63-0] :
                   (xmm1[63-0] > xmm2/m128[63-0]) ? xmm1[63-0] ? xmm2/m128[63-0];
xmm1[127-64] =
              xmm1[127-64] == NAN) ? xmm2[127-64] :
              (xmm2[127-64] == NAN) ? xmm2[127-64] :
              (xmm1[127-64] > xmm2/m128[127-64]) ? xmm1[127-64] ? xmm2/m128[127-64];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	If CR0.EM = 1.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Invalid (including QNaN source operand), Denormal.

Comments

Note that if only one source is a NaN for these instructions, the Src2 operand (either NaN or real value) is written to the result; this differs from the behavior for other instructions as defined in Table 2-3, which is to always write the NaN to the result, regardless of which source operand contains the NaN. This approach for MAXPD allows compilers to use the MAXPD instruction for common C conditional constructs. If instead of this behavior, it is required that the NaN source operand be returned, the min./max functionality can be emulated using a sequence of instructions: comparison followed by AND, ANDN and OR.

MAXSD—Scalar Double-Precision Floating-Point Maximum

Instruction	Description
MAXSD xmm1, xmm2/m64	Return the maximum double-precision floating-point number between the low double-precision floating-point numbers from xmm2/mem64 and xmm1.

Description

Returns the maximum double-precision floating-point number from the low double-precision floating-point numbers of xmm1 and xmm2/mem64. If the values being compared are both zeros, source operand 2 (xmm2/m128) is returned. If source operand 2 is an SNaN, that SNaN is forwarded unchanged to the destination (i.e., a quieted version of the SNaN is not returned).

Operation

```
xmm1[63-0]   =
                (xmm1[63-0] == NAN) ? xmm2[63-0] :
                (xmm2[63-0] == NAN) ? xmm2[63-0] :
                (xmm1[63-0] > xmm2/m64[63-0]) ? xmm1[63-0] ? xmm2/m64[63-0];
xmm1[127-64] = xmm1[127-64];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Invalid (including QNaN source operand), Denormal.

Comments

Note that if only one source is a NaN for these instructions, the Src2 operand (either NaN or real value) is written to the result; this differs from the behavior for other instructions as defined in Table 2-3, which is to always write the NaN to the result, regardless of which source operand contains the NaN. The upper three operands are still bypassed from the src1 operand, as in all other scalar operations. This approach for MAXSD allows compilers to use the MAXSD instruction for common C conditional constructs. If instead of this behavior, it is required that the NaN source operand be returned, the min/max functionality can be emulated using a sequence of instructions: comparison followed by AND, ANDN and OR.

MINPD—Packed Double-Precision Floating-Point Minimum

Instruction	Description
MINPD xmm1, xmm2/m128	Return the minimum double-precision floating-point numbers between xmm2/mem128 and xmm1.

Description

Returns the minimum double-precision floating-point numbers from the pairs of elements in xmm1 and xmm2/mem128. If the values being compared are both zeros, source operand 2 (xmm2/m128) is returned. If a value in source operand 2 is an SNaN, that SNaN is forwarded unchanged to the destination (i.e., a quieted version of the SNaN is not returned).

Operation

```
xmm1[63-0]   =
            (xmm1[63-0] == NAN) ? xmm2[63-0] :
            (xmm2[63-0] == NAN) ? xmm2[63-0] :
            (xmm1[63-0] < xmm2/m128[63-0]) ? xmm1[63-0] ? xmm2/m128[63-0];
xmm1[127-64] =
            (xmm1[127-64] == NAN) ? xmm2[127-64] :
            (xmm2[127-64] == NAN) ? xmm2[127-64] :
            (xmm1[127-64] < xmm2/m128[127-64]) ? xmm1[127-64] ? xmm2/m128[127-64];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	If CR0.EM = 1.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Invalid (including QNaN source operand), Denormal.

Comments

Note that if only one source is a NaN for these instructions, the Src2 operand (either NaN or real value) is written to the result; this differs from the behavior for other instructions as defined in Table 2-3, which is to always write the NaN to the result, regardless of which source operand contains the NaN. This approach for MINPD allows compilers to use the MINPD instruction for common C conditional constructs. If instead of this behavior, it is required that the NaN source operand be returned, the min/max functionality can be emulated using a sequence of instructions: comparison followed by AND, ANDN and OR.

MINSD—Scalar Double-Precision Floating-Point Minimum

Instruction	Description
MINSD xmm1, xmm2/m64	Return the minimum double-precision floating-point number between the low double-precision floating-point numbers from xmm2/mem64 and xmm1.

Description

Returns the minimum double-precision floating-point number from the low double-precision floating-point numbers of xmm1 and xmm2/mem64. If the values being compared are both zeros, source operand 2 (xmm2/m128) is returned. If source operand 2 is an SNaN, that SNaN is forwarded unchanged to the destination (i.e., a quieted version of the SNaN is not returned).

Operation

```
xmm1[63-0]   =
                    (xmm1[63-0] == NAN) ? xmm2[63-0] :
                    (xmm2[63-0] == NAN) ? xmm2[63-0] :
                    (xmm1[63-0] < xmm2/m64[63-0]) ? xmm1[63-0] ? xmm2/m64[63-0];
xmm1[127-64] = xmm1[127-64];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Invalid (including QNaN source operand), Denormal.

Comments

Note that if only one source is a NaN for these instructions, the Src2 operand (either NaN or real value) is written to the result (if SNaN, not converted to QNaN, but forwarded intact); this differs from the behavior for other instructions as defined in Table 2-3, which is to always write the NaN to the result, regardless of which source operand contains the NaN. This approach for MINSD allows NaN data to be screened out of the bounds-checking portion of an algorithm. If instead of this behavior, it is required that the NaN source operand be returned, the min/max functionality can be emulated using a sequence of instructions: comparison followed by AND, ANDN and OR.

MOVAPD—Move Aligned Two Packed Double-Precision Floating-Point

Instruction	Description
MOVAPD xmm1, xmm2/m128	Move packed double-precision floating-point numbers from xmm2/mem128 to xmm1.
MOVAPD xmm2/m128, xmm1	Move packed double-precision floating-point numbers from xmm1 to xmm2/mem128.

Description

Moves a double quadword containing two packed double-precision floating-point numbers from the source operand to the destination operand. When the source or destination operand is memory operand, the linear address of the operand is the address of the least-significant byte of the referenced data.

Operation

```
if (destination == xmm1) {
    if (source == m128) {
         // load instruction
         xmm1[127-0] = m128;
    }
    else {
         // move instruction
         xmm1[127=0] = xmm2[127-0];
    }
}
else {
    if (destination == m128) {
         // store instruction
         m128 = xmm1[127-0];
    }
    else {
         // move instruction
         xmm2[127-0] = xmm1[127-0];
    }
}
```

Protected Mode Exceptions

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

#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

Comments

The MOVAPD instruction should be used when dealing with 16-byte aligned double-precision floating-point numbers. If the data is not known to be aligned, the MOVUPD instruction should be used instead of MOVAPD. The use of this instruction should be limited to the cases where the aligned restriction is easy to meet. Processors that support Streaming SIMD Extensions 2 instructions technology will provide optimal aligned performance for the MOVAPD instruction.

MOVHPD—Move High Packed Double-Precision Floating-Point

Instruction	Description
MOVHPD xmm, m64	Move double-precision floating-point number from memory to high quadword of XMM register.
MOVHPD m64, xmm	Move double-precision floating-point number from high quadword of XMM register to memory.

Description

Moves a quadword containing a double-precision floating-point number from the source operand to the destination operand. When the source or destination operand is xmm, the high quadword of the register is affected. When the source or destination operand is memory operand, the linear address of the operand is the address of the least-significant byte of the referenced data.

Operation

```
if (destination == xmm) {
    // load instruction
    xmm[63-0]  = xmm[63-0];
    xmm[127-64]   = m64;
}
else {
    // store instruction
    m64 = xmm[127-64];
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.

#UD	If CR0.EM = 1.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Invalid (including QNaN source operand), Denormal.

Comments

The use of Repeat Prefixes (F2H, F3H) with MOVHPD is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with MOVHPD risks incompatibility with future processors.

MOVLPD—Move Low Packed Double-Precision Floating-Point

Instruction	Description
MOVLPD xmm, m64	Move double-precision floating-point numbers from memory to low quadword of XMM register.
MOVLPD m64, xmm	Move double-precision floating-point numbers from low quadword of XMM register to memory.

Description

Moves a quadword containing a double-precision floating-point number from the source operand to the destination operand. When the source or destination operand is xmm, the low quadword of the register is affected. When the source or destination operand is memory operand, the linear address of the operand is the address of the least-significant byte of the referenced data.

Operation

```
if (destination == xmm) {
    // load instruction
    xmm[63-0]   = m64;
    xmm[127-64] = xmm[127-64];
}
else {
    // store instruction
    m64 = xmm[63-0];
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

None.

Comments

The use of Repeat Prefixes (F2H, F3H) with MOVLPD is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with MOVLPD risks incompatibility with future processors.

MOVMSKPD—Move Mask To Integer

Instruction	Description
MOVMSKPD r32, xmm	Move 2-bit mask to r32.

Description

Moves a 2-bit mask containing the sign bits of the two packed double-precision floating-point numbers in xmm to a general-purpose register.

Operation

```
r32[0]          = xmm[63];
r32[1]          = xmm[127];
r32[7-2]        = 0x00;
r32[15-8]       = 0x00;
r32[31-16]      = 0x0000;
```

Protected Mode Exceptions

#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

Numeric Exceptions

None.

MOVSD—Move Scalar Double-Precision Floating-Point

Instruction	Description
MOVSD xmm1, xmm2/m64	Move scalar double-precision floating-point numbers from xmm2/m64 to xmm1 register.
MOVSD xmm2/m64, xmm	Move scalar double-precision floating-point numbers from xmm1 register to xmm2/m64.

Description

Moves a quadword containing a double-precision floating-point number from the source operand to the destination operand. When the source or destination operand is xmm, the low quadword of the register is affected. When the source or destination operand is memory operand, the linear address of the operand is the address of the least-significant byte of the referenced data.

Operation

```
if (destination == xmm1) {
    if (source == m64) {
        // load instruction
        xmm1[63-0]   = m64;
        xmm1[127-64]  = 0x00000000;
    }
    else {
        // move instruction
        xmm1[63-0]   = xmm2[63-0];
        xmm1[127-64]  = xmm1[127-64];
    }
}
else {
    if (destination == m64) {
        // store instruction
        m64 = xmm1[63-0];
    }
    else {
        // move instruction
        xmm2[63-0]   = xmm1[63-0]
        xmm2[127-64]  = xmm2[127-64];
    }
}
```

Protected Mode Exceptions

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

#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

None.

MOVUPD—Move Unaligned Two Packed Double-Precision Floating-Point

Instruction	Description
MOVUPD xmm1, xmm2/m1281	Move two packed double-precision floating-point numbers from xmm2/mem128 to xmm1.
MOVUPD xmm2/m128, xmm	Move two packed double-precision floating-point numbers from xmm1 to xmm2/mem128.

Description

Moves a double quadword containing two packed double-precision floating-point numbers from the source operand to the destination operand. When the source or destination operand is memory operand, the linear address of the operand is the address of the least-significant byte of the referenced data, and no assumption is made about alignment. That is, unaligned memory accesses can be made without generating a general-protection (#GP) exception.

Operation

```
if (destination == xmm1) {
    if (source == m128) {
        // load instruction
        xmm1[127-0] = m128;
    }
    else {
        // move instruction
        xmm1[127-0] = xmm2[127-0];
    }
}
else {
    if (destination == m128) {
        // store instruction
        m128 = xmm1[127-0];
    }
    else {
        // move instruction
        xmm2[127-0] = xmm1[127-0];
    }
}
```

Protected Mode Exceptions

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

#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

Comments.

The MOVUPD instruction should be used with double-precision floating-point numbers when that data is known to be unaligned. The use of this instruction should be limited to the cases where the aligned restriction is hard or impossible to meet. Streaming SIMD Extensions 2 implementations guarantee optimum unaligned support for MOVUPD. Efficient Streaming SIMD Extensions applications should mainly rely on MOVAPD, not MOVUPD, when dealing with aligned data.

The use of Repeat-NE Prefix (F2H) and Operand Size Prefix (66H) with MOVUPD is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with MOVUPD risks incompatibility with future processors.

A linear address of the 128-bit data access, while executing in 16-bit mode, that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. Different processor implementations may/may not raise a GP fault in this case if the segment limit has been exceeded; additionally, the address that spans the end of the segment may/may not wrap around to the beginning of the segment.

MULPD—Packed Double-Precision Floating-Point Multiply

Instruction	Description
MULPD xmm1, xmm2/m128	Multiply packed double-precision floating-point numbers in xmm2/mem128 by xmm1.

Description

Multiplies the packed double-precision floating-point numbers in xmm2/mem128 by the packed double-precision floating-point numbers in xmm1, and returns the packed result to xmm1.

Operation

```
xmm1[63-0]    = xmm1[63-0] * xmm2/m128[63-0];
xmm1[127-64]  = xmm1[127-64] * xmm2/m128[127-64];
```

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).

#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

MULSD—Scalar Double-Precision Floating-Point Multiply

Instruction	Description
MULSD xmm1 xmm2/m64	Multiply the low double-precision floating-point number in xmm2/mem64 by low double-precision floating-point number in xmm1.

Description

Multiplies the low double-precision floating-point number in xmm2 or the double-precision floating-point number in mem64 by the low double-precision floating-point number in xmm1, and returns the result to the low quadword of xmm1.

Operation

```
xmm1[63-0]   = xmm1[63-0] * xmm2/m64[63-0];
xmm1[127-64] = xmm1[127-64];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).

#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

ORPD—Bitwise Logical OR for Double-Precision Floating-Point Data

Instruction	Description
ORPD xmm1, xmm2/m128	Bitwise OR of xmm2/mem128 and xmm1.

Description

Performs a bitwise logical OR of the double quadwords in xmm1 and xmm2/mem128, and returns the result to xmm1.

Operation

```
xmm1[127-0] |= xmm2/m128[127-0];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.

#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

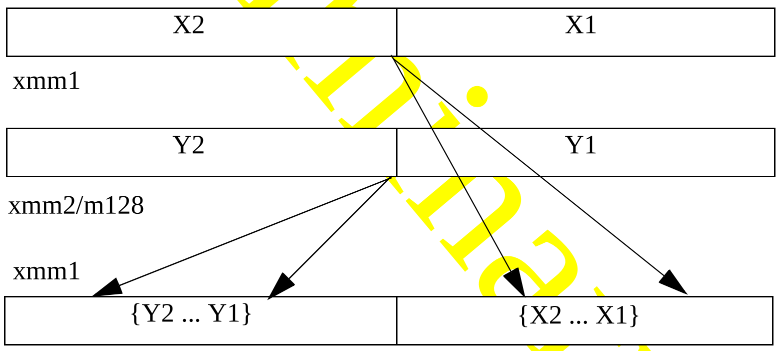
None.

SHUFPD—Shuffle Double-Precision Floating-Point

Instruction	Description
SHUFPD xmm1, xmm2/m128, imm8	Shuffle packed double-precision floating-point numbers.

Description

Shuffles either of the two packed double-precision floating-point numbers from xmm1 to the low quadword of xmm1; shuffles either of the two packed double-precision floating-point numbers from xmm2/m128 to the high quadword of xmm1. Bit 0 of the immediate field selects which of the two input double-precision floating-point numbers will be put in the low quadword of the result; bit 1 selects which of the two input double-precision floating-point numbers will be put in the high quadword of the result.



Operation

```
fp_select  = (imm8 >> 0) & 0x1;
xmm1[63-0]  =           (fp_select == 0) ? xmm1[63-0]   :
                        (fp_select == 1) ? xmm1[127-64];
fp_select  = (imm8 >> 1) & 0x1;
xmm1[127-64]  =         (fp_select == 0) ? xmm2/m128[63-0]   :
                        (fp_select == 1) ? xmm2/m128[127-64];
```

Protected Mode Exceptions

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

#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

SQRTPD—Packed Double-Precision Floating-Point Square Root

Instruction	Description
SQRTPD xmm1, xmm2/m128	Computes square roots of the packed double-precision floating-point numbers in xmm2/mem128.

Description

Computes the square roots of the two packed double-precision floating-point numbers from xmm2/m128 and returns the packed result in xmm1.

Operation

```
xmm1[63-0]   = sqrt (xmm2/m128[63-0]);
xmm1[127-64] = sqrt (xmm2/m128[127-64]);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.

#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

Invalid, Precision, Denormal.

SQRTSD—Scalar Double-Precision Floating-Point Square Root

Instruction	Description
SQRTSD xmm1, xmm2/m64	Computes square root of the low double-precision floating-point number in xmm2/mem64.

Description

Returns the square root of the low double-precision floating-point number in xmm or the double-precision floating-point number from memory and returns the result to xmm1. The high quadword of xmm1 is left unchanged.

Operation

```
xmm1[63-0]   = sqrt (xmm2/m64[63-0]);
xmm1[127-64] = xmm1[127-64];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).

#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Invalid, Precision, Denormal.

SUBPD—Packed Double-Precision Floating-Point Subtract

Instruction	Description
SUBPD xmm1 xmm2/m128	Subtract packed double-precision floating-point numbers in xmm2/mem128 from xmm1.

Description

Subtracts the two packed double-precision floating-point numbers in xmm2/mem128 from the two packed double-precision floating-point numbers in xmm1, and returns the results to xmm1.

Operation

```
xmm1[63-0]   = xmm1[63-0]   - xmm2/m128[63-0];
xmm1[127-64] = xmm1[127-64] - xmm2/m128[127-64];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.

#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

SUBSD—Scalar Double-Precision Floating-Point Subtract

Instruction	Description
SUBSD xmm1, xmm2/m64	Subtracts the low double-precision floating-point numbers in xmm2/mem64 from xmm1.

Description

Subtracts the low double-precision floating-point number in xmm2 or the double-precision floating-point number in mem64 from the low double-precision floating-point number in xmm1, and returns the result to the low quadword of xmm1.

Operation

```
xmm1[63-0]   = xmm1[63-0] - xmm2/m64[63-0];
xmm1[127-64] = xmm1[127-64];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).

#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

UCOMISD—Unordered Scalar Double-Precision Floating-Point Compare and Set EFLAGS

Instruction	Description
UCOMISD xmm1, xmm2/m64	Compares (unordered) the low double-precision floating-point number in xmm1 register with the low double-precision floating-point number in xmm2/mem64 and set the EFLAGS register accordingly.

Description

Performs an unordered compare of the low double-precision floating-point number in xmm2 or the double-precision floating-point number in mem64 with the low double-precision floating-point number in xmm1, and sets the ZF, PF, and CF flags in the EFLAGS register according to the result. The OF, SF and AF flags of the EFLAGS register are cleared. The unordered predicate is returned if either source operand is a NaN (QNaN or SNaN).

Operation

```
switch (xmm1[63-0] <> xmm2/m64[63-0]) {
    case UNORDERED:     ZF,PF,CF = 111;
    case GREATER_THAN:  ZF,PF,CF = 000;
    case LESS_THAN:     ZF,PF,CF = 001;
    case EQUAL:         ZF,PF,CF = 100;
  OF,SF,AF = 0;
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

Invalid (if SNaN operands), Denormal. Integer EFLAGS values will not be updated in the presence of unmasked numeric exceptions.

Comments

The UCOMISD instruction differs from the COMISD instruction in that it signals an invalid numeric exception when a source operand is an SNaN; COMISD signals invalid if a source operand is either a QNaN or an SNaN.

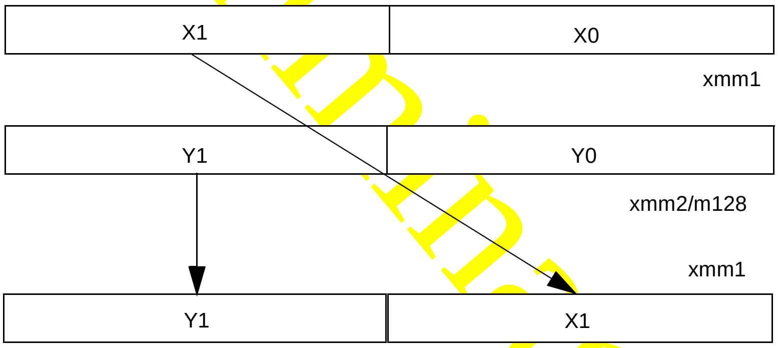
The use of Repeat (F2H, F3H) and Operand-Size prefixes with UCOMISD is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with UCOMISD risks incompatibility with future processors.

UNPCKHPD—Unpack High Packed Double-Precision Floating-Point Data

Instruction	Description
UNPCKHPD xmm1, xmm2/m128	Interleaves double-precision floating-point numbers from the high quadwords of xmm1 and xmm2/mem128.

Description

Performs an interleaved unpack of the high quadwords of xmm1 and xmm2/mem128. It ignores the low-order doublewords of the source operands.



When unpacking from a memory operand, an implementation may decide to fetch only the appropriate 64 bits. Alignment to 16-byte boundary and normal segment checking will still be enforced

Operation

```
xmm1[63-0]   = xmm1[127-64];
xmm1[127-64]  = xmm2/m128[127-64];
```

Protected Mode Exceptions

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

#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

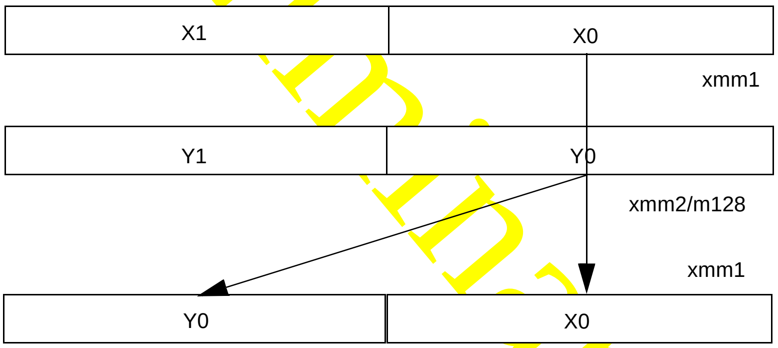
None

UNPCKLPD—Unpack Low Packed Double-Precision Floating-Point Data

Instruction	Description
UNPCKLPD xmm1, xmm2/m128	Interleaves double-precision floating-point numbers from the low quadwords of xmm1 and xmm2/mem128.

Description

Performs an interleaved unpack of the low quadwords of xmm1 and xmm2/mem128. It ignores the high element of the sources. When unpacking from a memory operand, the full 128-bit operand is accessed from memory but only the low order 64 bits are used by the instruction.



When unpacking from a memory operand, an implementation may decide to fetch only the appropriate 64 bits. Alignment to 16-byte boundary and normal segment checking will still be enforced

Operation

```
xmm1[63-0]   = xmm1[63-0];
xmm1[127-64]  = xmm2/m128[63-0];
```

Protected Mode Exceptions

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

#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None

XORPD—Bitwise Logical XOR for Double-Precision Floating-Point Data

Instruction	Description
XORPD xmm1, xmm2/m128	Bitwise XOR of xmm2/mem128 and xmm1.

Description

Performs a bitwise logical XOR of the double quadwords in xmm1 and xmm2/mem128, and returns the result to xmm1.

Operation

```
xmm[127-0] ^= xmm/m128[127-0];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.

#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None

3.3. SIMD INTEGER INSTRUCTIONS

This section describes the new and extended SIMD integer instructions that have been introduced with the Streaming SIMD Extensions 2 to operate on 128-packed integers. See Section 2.4.2., “SIMD Integer Instruction Extensions” for a summary of these additions.

CVTDQ2PS: Packed Signed Doubleword Integer to Packed Single-Precision Floating-Point Conversion

Instruction	Description
CVTDQ2PS xmm1,xmm2/m128	Convert four signed doubleword integers to four single-precision floating-point numbers.

Description

Converts four packed signed doubleword integers to four packed single-precision floating-point numbers. When the conversion is inexact, rounding is performed according to the rounding control bits in the MXCSR register.

Operation

```
xmm1[31-0]   = (float) (xmm2/m128[31-0]);
xmm1[63-32]  = (float) (xmm2/m128[63-32]);
xmm1[95-64]  = (float) (xmm2/m128[95-64]);
xmm1[127-96]  = (float) (xmm2/m128[127-96]);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

Precision.

CVTPS2DQ—Packed Single-Precision Floating-Point to Packed Doubleword Integer Conversion

Instruction	Description
CVTPS2DQ xmm1, xmm2/m128	Convert four packed single-precision floating-point numbers from xmm2/m128 to four packed doubleword signed integers in xmm1 using rounding specified by MXCSR.

Description

Converts the four packed single-precision floating-point numbers in xmm2/m128 to four packed signed doubleword integers in xmm1. When a converted value is inexact, the value is rounded according to the rounding control bits in the MXCSR register. If the converted result in one or more integers is larger than the maximum signed doubleword integer, the indefinite integer value (80000000H) is returned.

Operation

```
xmm1[31-0]   = (int) (xmm2/m128[31-0]);
xmm1[63-32]  = (int) (xmm2/m128[63-32]);
xmm1[95-64]  = (int) (xmm2/m128[95-64]);
xmm1[127-96] = (int) (xmm2/m128[127-96]);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

Invalid, Precision.

CVTTPS2DQ—Packed Single-Precision Floating-Point to Packed Signed Doubleword Integer Conversion (Truncate)

Instruction	Description
CVTTPS2DQ xmm1, xmm2/m128	Convert four single-precision floating-point numbers from xmm2/m128 to four doubleword signed integers in xmm1 using truncate.

Description

Converts the four single-precision floating-point numbers in xmm2/m128 to four signed doubleword integers in xmm1; when the conversion is inexact, a truncated result is returned. If the converted result in one or more elements is larger than the maximum signed doubleword integer, the indefinite integer value (80000000H) is returned.

Operation

```
xmm1[31-0]   = (int) (xmm2/m128[31-0]);
xmm1[63-32]  = (int) (xmm2/m128[63-32]);
xmm1[95-64]  = (int) (xmm2/m128[95-64]);
xmm1[127-96]  = (int) (xmm2/m128[127-96]);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#XM	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =1).
#UD	For an unmasked Streaming SIMD Extensions 2 instructions numeric exception (CR4.OSXMMEXCPT =0).
	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

Invalid, Precision.

MOVD—Move Doubleword

Instruction	Description
MOVD xmm, r/m32	Move doubleword from general-purpose register or memory to XMM.
MOVD r/m32, xmm	Move doubleword from XMM register to general-purpose register/memory.

Description

Moves doubleword from the source operand to the destination operand. The destination and source operands can be either XMM registers, 32-bit memory operands, or general-purpose registers. The MOVD instruction cannot transfer data from an XMM register to an XMM register, from memory to memory, or from an general-purpose register to an general-purpose register.

When the destination operand is an XMM register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 128 bits. When the source operand is an XMM register, the low quadword of the register are written to a general-purpose register or a 32-bit memory location.

Operation

```
if (destination == xmm) {
    xmm[31-0]   = r/m32;
    xmm[63-32]  = 0x00000000;
    xmm[95-64]  = 0x00000000;
    xmm[127-96] = 0x00000000;
}
else {
    r/m32 = xmm[31-0];
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

None.

MOVDQA—Move Aligned Double Quadword

Instruction	Description
MOVDQA xmm1, xmm2/m128	Move aligned double quadword from xmm2/mem128 to xmm1.
MOVDQA xmm2/m128, xmm1	Move aligned double quadword from xmm1 to xmm2/mem128.

Description

Moves a double quadword from the source operand to the destination operand. When the source or destination operand is memory operand, the linear address of the operand is the address of the least-significant byte of the referenced data.

Operation

```
if (destination == xmm1) {
    if (source == m128){
         // load instruction
         xmm1[127-0] = m128;
    }
    else {
         // move instruction
         xmm1[127-0] = xmm2[127-0];
    }
}
else {
    if (destination == m128) {
         // store instruction
         m128 = xmm1[127-0];
    }
    else {
         // move instruction
         xmm2[127-0] = xmm1[127-0];
    }
}
```

Protected Mode Exceptions

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

#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

Comments

The MOVDQA instruction should be used when dealing with 16-byte aligned integer numbers. If the data is not known to be aligned, MOVDQU should be used instead of MOVDQA. The use of this instruction should be limited to the cases where the aligned restriction is easy to meet. Processors that support Streaming SIMD Extensions 2 instructions technology will provide optimal aligned performance for the MOVDQA instruction.

MOVDQU—Move Unaligned Double Quadword

Instruction	Description
MOVDQU xmm1, xmm2/m128	Move unaligned double quadword from xmm2/mem128 to xmm1.
MOVDQU xmm2/m128, xmm1	Move unaligned double quadword from xmm1 to xmm2/mem128.

Description

Moves a double quadword from the source operand to the destination operand. When the source or destination operand is memory operand, the linear address of the operand is the address of the least-significant byte of the referenced data. No assumption is made about alignment.

Operation

```
if (destination == xmm1){
    if (source == m128) {
        // load instruction
        xmm1[127-0] = m128;
    }
    else {
        // move instruction
        xmm1[127-0] = xmm2[127-0];
    }
}
else {
    if (destination == m128) {
        // store instruction
        m128 = xmm1[127-0];
    }
    else {
        // move instruction
        xmm2[127-0] = xmm1[127-0];
    }
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.

	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

Comments

The MOVDQU instruction should be used with 128-bit integer data when that data is known to be unaligned. The use of this instruction should be limited to the cases where the aligned restriction is hard or impossible to meet. Streaming SIMD Extensions 2 instructions technology implementations guarantee optimum unaligned support for MOVDQU. Efficient Streaming SIMD Extensions 2 instructions technology applications should mainly rely on the MOVDQA instruction, not MOVDQU, when dealing with aligned data.

MOVDQ2Q—Move Quadword

Instruction	Description
MOVDQ2Q mm, xmm	Move low quadword from XMM to MMX register.

Description

Moves the low quadword from an XMM register to an MMX register.

Operation

```
mm[63-0]   = xmm[63-0]
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

None.

Comments

This instruction behaves identically to an MMX instruction, in the presence of x87 FPU instructions:

- Transition from x87 FPU to MMX technology (TOS=0, FP valid bits set to all valid). However, the use of a memory source operand with this instruction will not result in a transition from x87 FPU to MMX technology.
- In the case of a Streaming SIMD Extensions numeric exception, the x87 FPU to MMX technology transition will occur before the numeric exception is signaled.
- If there is an existing #MF fault, that will take priority over the Streaming SIMD Extensions numeric exception and the x87 FPU to MMX technology transition.

MOVQ2DQ—Move Quadword

Instruction	Description
MOVQ2DQ xmm, mm	Move quadword from MMX register to low quadword of XMM.

Description

Moves the quadword from an MMX register to the low quadword of an XXM register.

Operation

```
xmm[63-0]   = mm[63-0];
xmm[127-64] = 0x00000000;
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

None.

Comments

This instruction behaves identically to an MMX instruction, in the presence of x87 FPU instructions:

- When there is a transition from x87 FPU to MMX technology (TOS=0, FP valid bits set to all valid). However, the use of a memory source operand with this instruction will not result in a transition from x87 FPU to MMX technology.
- If in the case of a Streaming SIMD Extensions numeric exception, the x87 FPU to MMX technology transition will occur before the numeric exception is signaled.
- If there is an existing #MF fault, that will take priority over the Streaming SIMD Extensions numeric exception and the x87 FPU to MMX technology transition.

MOVQ—Move Quadword

Instruction	Description
MOVQ xmm1, xmm2/m64	Move quadword from xmm2/mem64 to xmm1.
MOVQ xmm2/m64, xmm1	Move quadword from xmm1 to xmm2/mem64.

Description

Moves a quadword from the source operand to the destination operand. When the source or destination operand is memory operand, the linear address of the operand is the address of the least-significant byte of the referenced data. When the source operand is an XMM register, the low quadword is moved; when the destination operand is an XMM register, the quadword is stored to the low quadword of the register, and the high quadword is cleared.

Operation

```
if (destination == xmm1) {
    if (source == m64){
         // load instruction
         xmm1[63-0]   = m64;
      xmm1[127-64] = 0;
    }
    else {
         // move instruction
         xmm1[63-0]   = xmm2[63-0];
      xmm1[127-64] = 0;
    }
}
else {
    if (destination == m64) {
         // store instruction
         m64 = xmm1[63-0];
    }
    else {
         // move instruction
         xmm2[63-0] = xmm1[63-0];
         xmm2[127-64] = 0;
    }
}
```

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.

#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PACKSSWB/PACKSSDW—Pack with Signed Saturation

Instruction	Description
PACKSSWB xmm1, xmm2/m128	Pack signed words from xmm1 and xmm2/mem128 into signed bytes in xmm1, with signed saturation.
PACKSSDW xmm1, xmm2/m128	Pack signed doublewords from and xmm2/mem128 into signed words in xmm1, with signed saturation.

Description

Packs with signed saturation the signed data elements from the source and the destination operands and writes the results to the destination operand. The destination operand is an XMM register. The source operand can either be an XMM register or a 128-bit memory operand.

The PACKSSWB instruction packs eight signed words from the source operand and eight signed words from the destination operand into sixteen signed bytes in the destination register. If the signed value of a word is larger or smaller than the range of a signed byte, the value is saturated (in the case of an overflow to 7FH, in the case of an underflow to 80H).

The PACKSSDW instruction packs four signed doublewords from the source operand and four signed doublewords from the destination operand into eight signed words in the destination register. If the signed value of a doubleword is larger or smaller than the range of a signed word, the value is saturated (in the case of an overflow to 7FFFH, in the case of an underflow to 8000H).

Operation

```
if (instruction == PACKSSWB) {
    xmm1[7-0]     = SaturateSignedWordToSignedByte (xmm1[15-0]);
    xmm1[15-8]    = SaturateSignedWordToSignedByte (xmm1[31-16]);
    xmm1[23-16]   = SaturateSignedWordToSignedByte (xmm1[47-32]);
    xmm1[31-24]   = SaturateSignedWordToSignedByte (xmm1[63-48]);
    xmm1[39-32]   = SaturateSignedWordToSignedByte (xmm1[79-64]);
    xmm1[47-40]   = SaturateSignedWordToSignedByte (xmm1[95-80]);
    xmm1[55-48]   = SaturateSignedWordToSignedByte (xmm1[111-96]);
    xmm1[63-56]   = SaturateSignedWordToSignedByte (xmm1[127-112]);
    xmm1[71-64]   = SaturateSignedWordToSignedByte (xmm2/m128[15-0]);
    xmm1[79-72]   = SaturateSignedWordToSignedByte (xmm2/m128[31-16]);
    xmm1[87-80]   = SaturateSignedWordToSignedByte (xmm2/m128[47-32]);
    xmm1[95-88]   = SaturateSignedWordToSignedByte (xmm2/m128[63-48]);
    xmm1[103-96]  = SaturateSignedWordToSignedByte (xmm2/m128[79-64]);
    xmm1[111-104] = SaturateSignedWordToSignedByte (xmm2/m128[95-80]);
    xmm1[119-112] = SaturateSignedWordToSignedByte (xmm2/m128[111-96]);
    xmm1[127-120] = SaturateSignedWordToSignedByte (xmm2/m128[127-112]);
}
else if (instruction == PACKSSDW) {
    xmm1[15-0]    = SaturateSignedDwordToSignedWord (xmm1[31-0]);
    xmm1[31-16]   = SaturateSignedDwordToSignedWord (xmm1[63-32]);
    xmm1[47-32]   = SaturateSignedDwordToSignedWord (xmm1[95-64]);
    xmm1[63-48]   = SaturateSignedDwordToSignedWord (xmm1[127-96]);
    xmm1[79-64]   = SaturateSignedDwordToSignedWord (xmm2/m128[31-0]);
    xmm1[95-80]   = SaturateSignedDwordToSignedWord (xmm2/m128[63-32]);
    xmm1[111-96]  = SaturateSignedDwordToSignedWord (xmm2/m128[95-64]);
    xmm1[127-112] = SaturateSignedDwordToSignedWord (xmm2/m128[127-96]);
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PACKUSWB—Pack with Unsigned Saturation

Instruction	Description
PACKUSWB xmm1, xmm2/m128	Pack and saturate signed words from xmm1 and xmm2/mem128 into unsigned bytes in xmm1.

Description

Packs eight signed words from the source operand xmm2/m128 and eight signed words from the destination operand xmm1 into sixteen unsigned bytes in the destination register xmm1. If the signed value of a word is larger or smaller than the range of an unsigned byte, the value is saturated (in the case of an overflow to FFH, in the case of an underflow to 00H). The destination operand is an XMM register. The source operand can either be an XMM register or a 128-bit memory operand.

Operation

```
xmm1[7-0]     = SaturateSignedWordToUnsignedByte (xmm1[15-0]);
xmm1[15-8]    = SaturateSignedWordToUnsignedByte (xmm1[31-16]);
xmm1[23-16]   = SaturateSignedWordToUnsignedByte (xmm1[47-32]);
xmm1[31-24]   = SaturateSignedWordToUnsignedByte (xmm1[63-48]);
xmm1[39-32]   = SaturateSignedWordToUnsignedByte (xmm1[79-64]);
xmm1[47-40]   = SaturateSignedWordToUnsignedByte (xmm1[95-80]);
xmm1[55-48]   = SaturateSignedWordToUnsignedByte (xmm1[111-96]);
xmm1[63-56]   = SaturateSignedWordToUnsignedByte (xmm1[127-112]);
xmm1[71-64]   = SaturateSignedWordToUnsignedByte (xmm2/m128[15-0]);
xmm1[79-72]   = SaturateSignedWordToUnsignedByte (xmm2/m128[31-16]);
xmm1[87-80]   = SaturateSignedWordToUnsignedByte (xmm2/m128[47-32]);
xmm1[95-88]   = SaturateSignedWordToUnsignedByte (xmm2/m128[63-48]);
xmm1[103-96]  = SaturateSignedWordToUnsignedByte (xmm2/m128[79-64]);
xmm1[111-104] = SaturateSignedWordToUnsignedByte (xmm2/m128[95-80]);
xmm1[119-112] = SaturateSignedWordToUnsignedByte (xmm2/m128[111-96]);
xmm1[127-120] = SaturateSignedWordToUnsignedByte (xmm2/m128[127-112]);
```

Protected Mode Exceptions

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

	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PADDB/PADDW/PADDD—Packed Add

Instruction	Description
PADDB xmm1,xmm2/m128	Add packed bytes from xmm2/mem128 to packed bytes in xmm1.
PADDW xmm1, xmm2/m128	Add packed words from xmm2/mem128 to packed words in xmm1.
PADDD xmm1, xmm2/m128	Add packed doublewords from xmm2/mem128 to packed doublewords in xmm1.

Description

Adds the packed data elements of the source operand to the packed data elements of the destination operand. The result is written to the destination operand. If the result is larger than the architectural limit, it wraps around. The destination operand is an XMM register. The source operand can either be an XMM register or a 128-bit memory operand.

The PADDB instruction adds the packed bytes from the source operand xmm2/m128 to the packed bytes in the destination operand xmm1 and writes the results to the xmm1 register. When the result is too large to be represented in a packed byte (overflow), the result wraps around and the low 8 bits are written to the destination register.

The PADDW instruction adds the packed words from the source operand xmm2/m128 to the packed words in the destination operand xmm1 and writes the results to the xmm1 register. When the result is too large to be represented in a packed word (overflow), the result wraps around and the low 16 bits are written to the destination register.

The PADDD instruction adds the packed doublewords from the source operand xmm2/m128 to the packed doublewords in the destination operand xmm1 and writes the results to the xmm1 register. When the result is too large to be represented in a packed doubleword (overflow), the result wraps around and the low 32 bits are written to the destination register.

Operation

```
if (instruction == PADDB) {
    xmm1[7-0]     = xmm1[7-0]     + xmm2/m128[7-0];
    xmm1[15-8]    = xmm1[15-8]    + xmm2/m128[15-8];
    xmm1[23-16]   = xmm1[23-16]   + xmm2/m128[23-16];
    xmm1[31-24]   = xmm1[31-24]   + xmm2/m128[31-24];
    xmm1[39-32]   = xmm1[39-32]   + xmm2/m128[39-32];
    xmm1[47-40]   = xmm1[47-40]   + xmm2/m128[47-40];
    xmm1[55-48]   = xmm1[55-48]   + xmm2/m128[55-48];
    xmm1[63-56]   = xmm1[63-56]   + xmm2/m128[63-56];
    xmm1[71-64]   = xmm1[71-64]   + xmm2/m128[71-64];
    xmm1[79-72]   = xmm1[79-72]   + xmm2/m128[79-72];
    xmm1[87-80]   = xmm1[87-80]   + xmm2/m128[87-80];
    xmm1[95-88]   = xmm1[95-88]   + xmm2/m128[95-88];
    xmm1[103-96]  = xmm1[103-96]  + xmm2/m128[103-96];
    xmm1[111-104] = xmm1[111-104] + xmm2/m128[111-104];
    xmm1[119-112] = xmm1[119-112] + xmm2/m128[119-112];
    xmm1[127-120] = xmm1[111-120] + xmm2/m128[127-120];
}
else if (instruction == PADDW){
    xmm1[15-0]    = xmm1[15-0]    + xmm2/m128[15-0];
    xmm1[31-16]   = xmm1[31-16]   + xmm2/m128[31-16];
    xmm1[47-32]   = xmm1[47-32]   + xmm2/m128[47-32];
    xmm1[63-48]   = xmm1[63-48]   + xmm2/m128[63-48];
    xmm1[79-64]   = xmm1[79-64]   + xmm2/m128[79-64];
    xmm1[95-80]   = xmm1[95-80]   + xmm2/m128[95-80];
    xmm1[111-96]  = xmm1[111-96]  + xmm2/m128[111-96];
    xmm1[127-112] = xmm1[127-112] + xmm2/m128[127-112];
}
else {
    // instruction is PADDD
    xmm1[31-0]  = xmm1[31-0]  + xmm2/m128[31-0];
    xmm1[63-32] = xmm1[63-32] + xmm2/m128[63-32];
    xmm1[95-64]  = xmm1[95-64]  + xmm2/m128[95-64];
    xmm1[127-96] = xmm1[127-96] + xmm2/m128[127-96];
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.

	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PADDQ—Packed Add Quadwords

Instruction	Description
PADDQ mm1,mm2/m64	Add quadword integers from mm2/Mem to mm1.

Description

Adds the quadword from the source operand (mm2/m64) to the quadword integer in the destination operand (mm1) and writes the result to the mm1 register. When the result is too large to be represented in a quadword (overflow), the result wraps around and the low 64 bits are written to the destination register (that is, the carry is ignored).

Note that like the integer ADD instruction, the PADDQ instruction can operate on either unsigned or signed (two's complement notation) integers. Unlike the integer instructions, none of the 64-bit or 128-bit integer instructions affect the EFLAGS register. With these integer instructions, there is no carry or overflow flags to indicate when overflow has occurred, so the software must control the range of values.

Operation

```
mm1[63-0] = mm1[63-0] + mm2/m64[63-0];
```

Exceptions: None.

Numeric Exceptions: None

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.

#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

None.

PADDQ—Packed Add Quadword 128 Bits

Instruction	Description
PADDQ xmm1,xmm2/m128	Add packed quadword integers from XMM2 /Mem to packed quadword integers in xmm1 register.

Description

Adds the packed quadword integers from the source operand (xmm2/m128) to the packed quadword integers in the destination operand (xmm1) and writes the results to the xmm1 register. When a quadword result is too large to be represented in a quadword element (overflow), the result wraps around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that like the integer ADD instruction, the PADDQ instruction can operate on either unsigned or signed (two’s complement notation) integers. Unlike the integer instructions, none of the 64-bit or 128-bit integer instructions affect the EFLAGS register. With these integer instructions, there is no carry or overflow flags to indicate when overflow has occurred, so the software must control the range of values.

Operation

```
xmm1[63-0] = xmm1[63-0] + xmm2/m128[63-0];
xmm1[127-64] = xmm1[127-64] + xmm2/m128[127-64];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PADDSB/PADDSW—Packed Add with Saturation

Instruction	Description
PADDSB xmm1, xmm2/m128	Add packed signed byte integers from xmm2/mem128 to packed signed bytes in xmm1, with saturation.
PADDSW xmm1, xmm2/m128	Add packed signed word integers from xmm2/mem128 to packed signed words in xmm1, with saturation.

Description

Adds the packed signed integer data elements of the source operand (xmm2/m128) to the packed signed integer data elements of the destination operand (xmm1) and saturates the result. The resulting packed signed integer elements are written to the destination operand (xmm1).

The PADDSB instruction operates on packed signed byte integers. If a result data element is larger or smaller than the range of a signed byte integer, the element is saturated to 7FH (in the case of an overflow) or to 80H (in the case of an underflow).

The PADDSB instruction operates on packed signed word integers. If a result data element is larger or smaller than the range of a signed word integer, the element is saturated to 7FFFH (in the case of an overflow) or to 8000H (in the case of an underflow).

Operation

```
if (instruction == PADDSB) {
    xmm1[7-0]     = SaturateToSignedByte (xmm1[7-0]     + xmm2/m128[7-0]);
    xmm1[15-8]    = SaturateToSignedByte (xmm1[15-8]    + xmm2/m128[15-8]);
    xmm1[23-16]   = SaturateToSignedByte (xmm1[23-16]   + xmm2/m128[23-16]);
    xmm1[31-24]   = SaturateToSignedByte (xmm1[31-24]   + xmm2/m128[31-24]);
    xmm1[39-32]   = SaturateToSignedByte (xmm1[39-32]   + xmm2/m128[39-32]);
    xmm1[47-40]   = SaturateToSignedByte (xmm1[47-40]   + xmm2/m128[47-40]);
    xmm1[55-48]   = SaturateToSignedByte (xmm1[55-48]   + xmm2/m128[55-48]);
    xmm1[63-56]   = SaturateToSignedByte (xmm1[63-56]   + xmm2/m128[63-56]);
    xmm1[71-64]   = SaturateToSignedByte (xmm1[71-64]   + xmm2/m128[71-64]);
    xmm1[79-72]   = SaturateToSignedByte (xmm1[79-72]   + xmm2/m128[79-72]);
    xmm1[87-80]   = SaturateToSignedByte (xmm1[87-80]   + xmm2/m128[87-80]);
    xmm1[95-88]   = SaturateToSignedByte (xmm1[95-88]   + xmm2/m128[95-88]);
    xmm1[103-96]  = SaturateToSignedByte (xmm1[103-96]  + xmm2/m128[103-96]);
    xmm1[111-104] = SaturateToSignedByte (xmm1[111-104] + xmm2/m128[111-104]);
    xmm1[119-112] = SaturateToSignedByte (xmm1[119-112] + xmm2/m128[119-112]);
    xmm1[127-120] = SaturateToSignedByte (xmm1[111-120] + xmm2/m128[127-120]);
}
else {
    // instruction is PADDW
    xmm1[15-0]    = SaturateToSignedWord (xmm1[15-0]    + xmm2/m128[15-0]);
    xmm1[31-16]   = SaturateToSignedWord (xmm1[31-16]   + xmm2/m128[31-16]);
    xmm1[47-32]   = SaturateToSignedWord (xmm1[47-32]   + xmm2/m128[47-32]);
    xmm1[63-48]   = SaturateToSignedWord (xmm1[63-48]   + xmm2/m128[63-48]);
    xmm1[79-64]   = SaturateToSignedWord (xmm1[79-64]   + xmm2/m128[79-64]);
    xmm1[95-80]   = SaturateToSignedWord (xmm1[95-80]   + xmm2/m128[95-80]);
    xmm1[111-96]  = SaturateToSignedWord (xmm1[111-96]  + xmm2/m128[111-96]);
    xmm1[127-112] = SaturateToSignedWord (xmm1[127-112] + xmm2/m128[127-112]);
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PADDUSB/PADDUSW-Packed Add Unsigned with Saturation

Instruction	Description
PADDUSB xmm1, xmm2/m128	Add packed unsigned byte integers from xmm2/mem128 to packed unsigned byte integers in xmm1, with saturation.
PADDUSW xmm1, xmm2/m128	Add packed unsigned word integers from xmm2/mem128 to packed unsigned word integers in xmm1, with saturation.

Description

Adds the packed unsigned integer data elements of the source operand (xmm2/m128) to the packed unsigned integer data elements of the destination operand (xmm1) and saturates the results. The results are written to the destination operand (xmm1).

The PADDUSB instruction operates on packed unsigned byte integers. If a result data element is larger or smaller than the range of an unsigned byte integer, the element is saturated to FFH (in the case of an overflow) or to 00H (in the case of an underflow).

The PADDUSW instruction operates on packed unsigned word integers. If a result data element is larger or smaller than the range of an unsigned word integer, the element is saturated to FFFFH (in the case of an overflow) or to 0000H (in the case of an underflow).

Operation

```
if (instruction == PADDUSB) {
    xmm1[7-0]     = SaturateToUnsignedByte (xmm1[7-0]     + xmm2/m128[7-0]);
    xmm1[15-8]    = SaturateToUnsignedByte (xmm1[15-8]    + xmm2/m128[15-8]);
    xmm1[23-16]   = SaturateToUnsignedByte (xmm1[23-16]   + xmm2/m128[23-16]);
    xmm1[31-24]   = SaturateToUnsignedByte (xmm1[31-24]   + xmm2/m128[31-24]);
    xmm1[39-32]   = SaturateToUnsignedByte (xmm1[39-32]   + xmm2/m128[39-32]);
    xmm1[47-40]   = SaturateToUnsignedByte (xmm1[47-40]   + xmm2/m128[47-40]);
    xmm1[55-48]   = SaturateToUnsignedByte (xmm1[55-48]   + xmm2/m128[55-48]);
    xmm1[63-56]   = SaturateToUnsignedByte (xmm1[63-56]   + xmm2/m128[63-56]);
    xmm1[71-64]   = SaturateToUnSignedByte (xmm1[71-64]   + xmm2/m128[71-64]);
    xmm1[79-72]   = SaturateToUnSignedByte (xmm1[79-72]   + xmm2/m128[79-72]);
    xmm1[87-80]   = SaturateToUnSignedByte (xmm1[87-80]   + xmm2/m128[87-80]);
    xmm1[95-88]   = SaturateToUnSignedByte (xmm1[95-88]   + xmm2/m128[95-88]);
    xmm1[103-96]  = SaturateToUnSignedByte (xmm1[103-96]  + xmm2/m128[103-96]);
    xmm1[111-104] = SaturateToUnSignedByte (xmm1[111-104] + xmm2/m128[111-104]);
    xmm1[119-112] = SaturateToUnSignedByte (xmm1[119-112] + xmm2/m128[119-112]);
    xmm1[127-120] = SaturateToUnSignedByte (xmm1[127-120] + xmm2/m128[127-120]);
}
else {
    // instruction is PADDUSW
    xmm1[15-0]    = SaturateToUnsignedWord (xmm1[15-0]    + xmm2/m128[15-0]);
    xmm1[31-16]   = SaturateToUnsignedWord (xmm1[31-16]   + xmm2/m128[31-16] ;
    xmm1[47-32]   = SaturateToUnsignedWord (xmm1[47-32]   + xmm2/m128[47-32]);
    xmm1[63-48]   = SaturateToUnsignedWord (xmm1[63-48]   + xmm2/m128[63-48]);
                                                                              )
    xmm1[79-64]   = SaturateToUnSignedWord (xmm1[79-64]   + xmm2/m128[79-64]);
    xmm1[95-80]   = SaturateToUnSignedWord (xmm1[95-80]   + xmm2/m128[95-80]);
    xmm1[111-96]  = SaturateToUnSignedWord (xmm1[111-96]  + xmm2/m128[111-96]);
    xmm1[127-112] = SaturateToUnSignedWord (xmm1[127-112] + xmm2/m128[127-112]);
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PAND—Bitwise Logical AND

Instruction	Description
PAND xmm1, xmm2/m128	Bitwise AND of xmm2/mem128 and xmm1.

Description

Performs a bitwise logical AND of the source operand (xmm/m128) and the destination operand (xmm1), and writes the result to the destination operand. Each bit of the result is a 1 if the corresponding bits of the operands are 1; otherwise, it is set to 0.

Operation

```
xmm1 = xmm1 & xmm2/m128;
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PANDN—Bitwise Logical And Not

Instruction	Description
PANDN xmm1, xmm2/m128	Bitwise AND NOT of xmm2/mem128 and xmm1.

Description

Performs a bitwise logical NOT on the destination operand, then performs a bitwise logical AND on the inverted destination operand and source operand. Each bit of the result of the AND instruction is a 1 if the corresponding bits are 1. Otherwise, it is set to 0. The result is written to the destination register. The destination operand is an XMM register. The source operand can either be an XMM register or a 128-bit memory operand.

Operation

```
xmm1 = (~xmm1) & xmm2/m128;
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PAVGB/PAVGW—Packed Average

Instruction	Description
PAVGB xmm1,xmm2/m128	Average packed unsigned bytes from xmm2/mem128 and xmm1, with rounding.
PAVGW xmm1, xmm2/m128	Average packed unsigned words from xmm2/mem128 and xmm1, with rounding.

Description

Adds the unsigned data elements of the source operand to the unsigned data elements of the destination operand, along with a carry-in. The resulting data elements are then each independently right-shifted by one bit position and the high-order bit of each element is filled with the carry bit of the corresponding sum. The destination operand is an XMM register. The source operand can either be an XMM register or a 128-bit memory operand.

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

Operation

```
if (instruction == PAVGB) {
    x[0]    = xmm1[7-0]          y[0] = xmm2/m128[7-0];
    x[1]    = xmm1[15-8]         y[1] = xmm2/m128[15-8];
    x[2]    = xmm1[23-16]        y[2] = xmm2/m128[23-16];
    x[3]    = xmm1[31-24]        y[3] = xmm2/m128[31-24];
    x[4]    = xmm1[39-32]        y[4] = xmm2/m128[39-32];
    x[5]    = xmm1[47-40]        y[5] = xmm2/m128[47-40];
    x[6]    = xmm1[55-48]        y[6] = xmm2/m128[55-48];
    x[7]    = xmm1[63-56]        y[7] = xmm2/m128[63-56];
    x[8]    = xmm1[71-64]        y[8] = xmm2/m128[71-64];
    x[9]    = xmm1[79-72]        y[9] = xmm2/m128[79-72];
    x[10]   = xmm1[87-80]        y[10] = xmm2/m128[87-80];
    x[11]   = xmm1[95-88]        y[11] = xmm2/m128[95-88];
    x[12]   = xmm1[103-96]       y[12] = xmm2/m128[103-96];
    x[13]   = xmm1[111-104]      y[13] = xmm2/m128[111-104];
    x[14]   = xmm1[119-112]      y[14] = xmm2/m128[119-112];
    x[15]   = xmm1[127-120]      y[15] = xmm2/m128[127-120];


    for (i = 0; i < 16; i++) {
        temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8);
        res[i] = (temp[i] +1) >> 1;
        }
    xmm1[7-0]       =   res[0];
    ...
    xmm1[127-120]   =   res[15];
```

```
        }
}
else if (instruction == PAVGW){
    x[0]      = xmm1[15-0]              y[0] = xmm2/m128[15-0];
    x[1]      = xmm1[31-16]             y[1] = xmm2/m128[31-16];
    x[2]      = xmm1[47-32]             y[2] = xmm2/m128[47-32];
    x[3]      = xmm1[63-48]             y[3] = xmm2/m128[63-48];
    x[4]      = xmm1[79-64]             y[4] = xmm2/m128[79-64];
    x[5]      = xmm1[95-80]             y[5] = xmm2/m128[95-80];
    x[6]      = xmm1[111-96]            y[6] = xmm2/m128[111-96];
    x[7]      = xmm1[127-112]           y[7] = xmm2/m128[127-112];

    for (i = 0; i < 8; i++) {
        temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16);
        res[i] = (temp[i] +1) >> 1;
    }
    xmm1[15-0]      =    res[0];
    ...
    xmm1[127-112]   =    res[7];
    }
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PCMPEQB/PCMPEQW/PCMPEQD—Packed Compare for Equal

Instruction	Description
PCMPEQB xmm1, xmm2/m128	Compare packed bytes in xmm2/mem128 to packed bytes in xmm1 for equality.
PCMPEQW xmm1, xmm2/m128	Compare packed words in xmm2/mem128 to packed words in xmm1 for equality.
PCMPEQD xmm1, xmm2/m128	Compare packed doublewords in xmm2/mem128 to packed doublewords in xmm1 for equality.

Description

Compares the data elements in the destination operand to the corresponding data elements in the source operand for equal values. If the data elements are equal, the corresponding data element in the destination register is set to all ones; if they are not equal, the corresponding data element is set to all zeros. The destination operand is an XMM register. The source operand can either be an XMM register or a 128-bit memory operand.

The PCMPEQB instruction compares the packed bytes in the destination operand to the packed bytes in the source operand.

The PCMPEQW instruction compares the packed words in the destination operand to the packed words in the source operand.

The PCMPEQD instruction compares the packed doublewords in the destination operand to the packed doublewords in the source operand.

Operation

```
if (instruction == PCMPEQB) {
    xmm1[7-0] = (xmm1[7-0]  == xmm2/m128[7-0])  ? 0xFF : 0x00;
    ...
    xmm1[127-120] = (xmm1[127-120] == xmm2/m128[127-120]) ? 0xFF : 0x00;
}
else if (instruction == PCMPEQW) {
    xmm1[15-0] = (xmm1[15-0] == xmm2/m128[15-0]) ? 0xFFFF : 0x0000;
    ...
    xmm1[127-112] = (xmm1[127-112] == xmm2/m128[127-112]) ? 0xFFFF : 0x0000;
}
else {
    // instruction is PCMPEQD
    xmm1[31-0] = (xmm1[31-0] == xmm2/m128[31-0]) ? 0xFFFFFFFF : 0x00000000;
         ...
         xmm1[127-96] = (xmm1[127-96] == xmm2/m128[127-96] ? 0xFFFFFFFF :
0x00000000;
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PCMPGTB/PCMPGTW/PCMPGTD—Packed Compare for Greater Than

Instruction	Description
PCMPGTB xmm1, xmm2/m128	Compare packed bytes in xmm1 with packed bytes in xmm2/mem128 for greater than.
PCMPGTW xmm1, xmm2/m128	Compare packed words in xmm1 with packed words in xmm2/mem128 for greater than.
PCMPGTD xmm1, xmm2/m128	Compare packed doublewords in xmm1 with packed doublewords in xmm2/mem128 for greater than.

Description

Compare the signed data elements in the destination operand to the corresponding signed data elements in the source operand. If the signed data elements in the destination register are greater than those in the source operand, the corresponding data element in the destination operand is set to all 1s, otherwise it is set to all 0s. The destination operand is an XMM register. The source operand can either be an XMM register or a 128-bit memory operand.

The PCMPGTB instruction compares the signed bytes in the destination operand to the corresponding signed bytes in the source operand.

he PCMPGTW instruction compares the signed words in the destination operand to the corresponding signed words in the source operand.

The PCMPGTD instruction compares the signed doublewords in the destination operand to the corresponding signed doublewords in the source operand.

Operation

```
if (instruction == PCMPGTB) {
    xmm1[7-0] = (xmm1[7-0] > xmm2/m128[7-0]) ? 0xFF : 0x00;
    ...
    xmm1[127-120] = (xmm1(127..120) > xmm2/m128[127-120]) ? 0xFF : 0x00;
}
else if (instruction == PCMPGTW) {
    xmm1[15-0] = (xmm1[15-0] > xmm2/m128[15-0]) ? 0xFFFF : 0x0000;
    ...
    xmm1[127-112] = (xmm1[127-112] > xmm2/m128[127-112]) ? 0xFFFF : 0x0000;
}
else {
    // instruction is PCMPGTD
    xmm1[32-0] = (xmm1[31-0] > xmm2/m128[31-0]) ? 0xFFFFFFFF : 0x00000000;
    ...
    xmm1[127-96] = (xmm1[127-96] > xmm2/m128[127-96]) ? 0xFFFFFFFF : 0x00000000;
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PEXTRW—Extract Word

Instruction	Description
PEXTRW r32, xmm, imm8	Extract the word specified by imm8 from XMM and move it to a general-purpose register.

Description

Moves the word in xmm selected by the three least-significant bits of imm8 to the low word of a general-purpose register.

Operation

```
sel = imm8 & 0x7;
xmm_temp = (xmm >> (sel * 16)) & 0xffff;
r[15-0] = xmm_temp[15-0];
r[31-16] = 0x0000;
```

Protected Mode Exceptions

#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

Numeric Exceptions

None.

PINSRW—Insert Word

Instruction	Description
PINSRW xmm, r32/m16, imm8	Move the low word of r32 or from m16 into xmm at the position specified by imm8.

Description

Moves the low word from r32 or a word from memory xmm at a position specified by the three least-significant bits of the immediate imm8. The seven other words from the destination register are left untouched.

Operation

```
sel = imm8 & 0x7;
mask = (sel == 0)?    0x0000000000000000000000000000ffff :
       (sel == 1)?    0x000000000000000000000000ffff0000 :
       (sel == 2)?    0x00000000000000000000ffff00000000 :
       (sel == 3)?    0x0000000000000000ffff000000000000;
       (sel == 4)?    0x000000000000ffff0000000000000000;
       (sel == 5)?    0x00000000ffff00000000000000000000;
       (sel == 6)?    0x0000ffff000000000000000000000000;
       (sel == 7)?    0xffff0000000000000000000000000000;
xmm = (xmm & ~mask) | ((m16/rNone.
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.

#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

None.

PMADDWD—Packed Multiply and Add

Instruction	Description
PMADDWD xmm1, xmm2/m128	Multiply the packed word integers in xmm1 by the packed word integers in xmm2/mem128, and add the adjacent doubleword results.

Description

Multiplies the eight signed word integers in the destination operand by the corresponding eight signed word integers in the source operand. The adjacent doubleword results are then summed and stored in the destination operand. For example, the two pairs of low-order words (15-0) and 31-16) are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. The result is four doublewords, stored in the destination operand.

The destination operand is an XMM register. The source operand can either be an XMM register or a 128-bit memory operand.

The PMADD instruction wraps around only in one situation, when the four words of both the source and destination operands are 8000H. In this case the result wraps around to 80000000H.

Operation

```
xmm1[31-0]   = xmm1[15-0]   * xmm2/m128[15-0]   + xmm1[31-16] * xmm2/m128[31-16];
xmm1[63-32]  = xmm1[47-32]  * xmm2/m128[47-32]  + xmm1[63-48] * xmm2/m128[63-48];
xmm1[95-64]  = xmm1[79-64]  * xmm2/m128[79-64]  + xmm1[95-80] * xmm2/m128[95-80];
xmm1[127-96] = xmm1[111-96] * xmm2/m128[111-96] + xmm1[127-112] * xmm2/m128[127-
112];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PMAXSW—Packed Signed Word Integer Maximum

Instruction	Description
PMAXSW xmm1, xmm2/m128	Return the maximum signed word integers between xmm2/mem128 and xmm1.

Description

Returns the maximum between each corresponding pair of signed word integers in xmm1 and xmm2/mem128.

Operation

```
xmm1[15-0]   = (xmm1[15-0] > xmm2/m128[15-0]) ? xmm1[15-0] : xmm2/m128[15-0];
xmm1[31-16]  = (xmm1[31-16] > xmm2/m128[31-16]) ? xmm1[31-16] : xmm2/m128[31-16];
xmm1[47-32]  = (xmm1[47-32] > xmm2/m128[47-32]) ? xmm1[47-32] : xmm2/m128[47-32];
xmm1[63-48]  = (xmm1[63-48] > xmm2/m128[63-48]) ? xmm1[63-48] : xmm2/m128[63-48];
xmm1[79-64]  = (xmm1[79-64] > xmm2/m128[31-16]) ? xmm1[79-64] : xmm2/m128[79-64];
xmm1[95-80]  = (xmm1[95-80] > xmm2/m128[47-32]) ? xmm1[95-80] : xmm2/m128[95-80];
xmm1[111-96] = (xmm1[111-96] > xmm2/m128[63-48]) ? xmm1[111-96] : xmm2/m128[111-96];
xmm1[127-112]= (xmm1[127-112] > xmm2/m128[63-48]) ? xmm1[127-112] : xmm2/m128[127-
112];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.

#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PMAXUB—Packed Unsigned Byte Integer Maximum

Instruction	Description
PMAXUB xmm1,xmm2/m128	Return the maximum unsigned byte integers between xmm2/m128 and xmm1.

Description

Returns the maximum between each corresponding pair of unsigned byte integers in xmm1 and xmm2/mem128.

Operation

```
xmm1[7-0]   = (xmm1[7-0] > xmm2/m128[7-0]) ?
              xmm1[7-0] : xmm2/m128[7-0];
xmm1[15-8]  = (xmm1[15-8] > xmm2/m128[15-8]) ?
              xmm1[15-8] : xmm2/m128[15-8];
xmm1[23-16]  = (xmm1[23-16] > xmm2/m128[23-16]) ?
              xmm1[23-16] : xmm2/m128[23-16];
xmm1[31-24]  = (xmm1[31-24] > xmm2/m128[31-24]) ?
                              xmm1[31-24] : xmm2/m128[31-24];
xmm1[39-32]  = (xmm1[39-32] > xmm2/m128[39-32]) ?
                              xmm1[39-32] : xmm2/m128[39-32];
xmm1[47-40]  = (xmm1[47-40] > xmm2/m128[47-40]) ?
                              xmm1[47-40] : xmm2/m128[47-40];
xmm1[55-48]  = (xmm1[55-48] > xmm2/m128[55-48]) ?
                              xmm1[55-48] : xmm2/m128[55-48];
xmm1[63-56]  = (xmm1[63-56] > xmm2/m128[63-56]) ?
                              xmm1[63-56] : xmm2/m128[63-56];
xmm1[71-64] = (xmm1[71-64] > xmm2/m128[71-64]) ?
                              xmm1[71-64] : xmm2/m128[71-64];
xmm1[79-72] = (xmm1[79-72] > xmm2/m128[79-72]) ?
                              xmm1[79-72] : xmm2/m128[79-72];
xmm1[87-80] = (xmm1[87-80] > xmm2/m128[87-80]) ?
                              xmm1[87-80] : xmm2/m128[87-80];
xmm1[95-88] = (xmm1[95-88] > xmm2/m128[95-88]) ?
                              xmm1[95-88] : xmm2/m128[95-88];
xmm1[103-96] = (xmm1[103-96] > xmm2/m128[103-96]) ?
                              xmm1[103-96] : xmm2/m128[103-96];
xmm1[111-104] = (xmm1[111-104] > xmm2/m128[111-104]) ?
                              xmm1[111-104] : xmm2/m128[111-104];
xmm1[119-112] = (xmm1[119-112] > xmm2/m128[119-112]) ?
                              xmm1[119-112] : xmm2/m128[119-112];
xmm1[127-120] = (xmm1[127-120] > xmm2/m128[127-120]) ?
                              xmm1[127-120] : xmm2/m128[127-120];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PMINSW—Packed Signed Integer Word Minimum

Instruction	Description
PMINSW xmm1, xmm2/m128	Return the minimum signed word integers between xmm2/mem128 and xmm1.

Description

Returns the minimum between each corresponding pair of signed word integers in xmm1 and xmm2/mem128.

Operation

```
xmm1[15-0]   = (xmm1[15-0] < xmm2/m128[15-0]) ? xmm1[15-0] : xmm2/m128[15-0];
xmm1[31-16]  = (xmm1[31-16] < xmm2/m128[31-16]) ? xmm1[31-16] : xmm2/m128[31-16];
xmm1[47-32]  = (xmm1[47-32] < xmm2/m128[47-32]) ? xmm1[47-32] : xmm2/m128[47-32];
xmm1[63-48]  = (xmm1[63-48] < xmm2/m128[63-48]) ? xmm1[63-48] : xmm2/m128[63-48];
xmm1[79-64]  = (xmm1[79-64] < xmm2/m128[31-16]) ? xmm1[79-64] : xmm2/m128[79-64];
xmm1[95-80]  = (xmm1[95-80] < xmm2/m128[47-32]) ? xmm1[95-80] : xmm2/m128[95-80];
xmm1[111-96] = (xmm1[111-96] < xmm2/m128[63-48]) ? xmm1[111-96] : xmm2/m128[111-96];
xmm1[127-112]= (xmm1[127-112] < xmm2/m128[63-48]) ? xmm1[127-112] : xmm2/m128[127-
112];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.

#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PMINUB—Packed Unsigned Byte Integer Minimum

Instruction	Description
PMINUB xmm1, xmm2/m128	Return the minimum unsigned byte integers between xmm2/mem128 and xmm1.

Description

Returns the minimum between each corresponding pair of unsigned byte integers in xmm1 and xmm2/mem128.

Operation

```
xmm1[7-0]   = (xmm1[7-0] < xmm2/m128[7-0]) ? xmm1[7-0] : xmm2/m128[7-0];
xmm1[15-8]  = (xmm1[15-8] < xmm2/m128[15-8]) ? xmm1[15-8] : xmm2/m128[15-8];
xmm1[23-16]  = (xmm1[23-16] < xmm2/m128[23-16]) ? xmm1[23-16] : xmm2/m128[23-16];
xmm1[31-24]  = (xmm1[31-24] < xmm2/m128[31-24]) ? xmm1[31-24] : xmm2/m128[31-24];
xmm1[39-32]  = (xmm1[39-32] < xmm2/m128[39-32]) ? xmm1[39-32] : xmm2/m128[39-32];
xmm1[47-40]  = (xmm1[47-40] < xmm2/m128[47-40]) ? xmm1[47-40] : xmm2/m128[47-40];
xmm1[55-48]  = (xmm1[55-48] < xmm2/m128[55-48]) ? xmm1[55-48] : xmm2/m128[55-48];
xmm1[63-56]  = (xmm1[63-56] < xmm2/m128[63-56]) ? xmm1[63-56] : xmm2/m128[63-56];
xmm1[71-64] = (xmm1[71-64] < xmm2/m128[71-64]) ? xmm1[71-64] : xmm2/m128[71-64];
xmm1[79-72] = (xmm1[79-72] < xmm2/m128[79-72]) ? xmm1[79-72] : xmm2/m128[79-72];
xmm1[87-80] = (xmm1[87-80] < xmm2/m128[87-80]) ? xmm1[87-80] : xmm2/m128[87-80];
xmm1[95-88] = (xmm1[95-88] < xmm2/m128[95-88]) ? xmm1[95-88] : xmm2/m128[95-88];
xmm1[103-96] = (xmm1[103-96] < xmm2/m128[103-96]) ? xmm1[103-96] : xmm2/m128[103-
96];
xmm1[111-104] = (xmm1[111-104] < xmm2/m128[111-104]) ? xmm1[111-104] :
xmm2/m128[111-104];
xmm1[119-112] = (xmm1[119-112] < xmm2/m128[119-112]) ? xmm1[119-112] :
xmm2/m128[119-112];
xmm1[127-120] = (xmm1[127-120] < xmm2/m128[127-120]) ? xmm1[127-120] :
xmm2/m128[127-120];
```

Protected Mode Exceptions

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

	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PMOVMSKB—Move Byte Mask To Integer

Instruction	Description
PMOVMSKB r32, xmm	Move the byte mask of xmm to r32.

Description

Returns a 16-bit mask formed of the most significant bits of each byte in xmm to the destination register r32.

Operation

```
r32[15]  = xmm[127];  r32[14]  = xmm[119];
r32[13]  = xmm[111];  r32[12]  = xmm[103];
r32[11]  = xmm[95];  r32[10]  = xmm[87];
r32[9]  = xmm[79];  r32[8]  = xmm[71];
r32[7]  = xmm[63];  r32[6]  = xmm[55];
r32[5]  = xmm[47];  r32[4]  = xmm[39];
r32[3]  = xmm[31];  r32[2]  = xmm[23];
r32[1]  = xmm[15];  r32[0]  = xmm[7];
r32[31-16] = 0x0000;
```

Protected Mode Exceptions

#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PMULHW—Packed Multiply High

Instruction	Description
PMULHW xmm1, xmm2/m128	Multiply the packed signed word integers in xmm1 by the packed signed word integers in xmm2/mem128, and store the high words of the results in xmm1.

Description

Multiplies the eight signed word integers in the destination operand by the corresponding eight signed word integers in the source operand, and store the high word of the individual double-word results in the destination operand. The destination operand is an XMM register. The source operand can either be an XMM register or a 128-bit memory operand.

Operation

```
xmm1[15-0]      = (xmm1[15-0]      * xmm2/m128[15-0])[31-16];
xmm1[31-16]     = (xmm1[31-16]     * xmm2/m128[31-16])[31-16];
xmm1[47-32]     = (xmm1[47-32]     * xmm2/m128[47-32])[31-16];
xmm1[63-48]     = (xmm1[63-48]     * xmm2/m128[63-48])[31-16];
xmm1[79-64]     = (xmm1[79-64]     * xmm2/m128[79-64])[31-16];
xmm1[95-80]     = (xmm1[95-80]     * xmm2/m128[95-80])[31-16];
xmm1[111-96]    = (xmm1[111-96]    * xmm2/m128[111-96])[31-16];
xmm1[127-112]   = (xmm1[127-112]   * emm2/m128[127-112])[31-16];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PMULHUW—Packed Multiply High Unsigned

Instruction	Description
PMULHUW xmm1, xmm2/m128	Multiply the packed unsigned word integers in xmm1 by the packed unsigned word integers in xmm2/mem128, and store the high words of the results in xmm1.

Description

Multiplies the eight unsigned word integers in the destination operand by the corresponding eight unsigned word integers in the source operand, and store the high word of the individual doubleword results in the destination operand.

Operation

```
xmm1[15-0]      = (xmm1[15-0]       * xmm2/m128[15-0])[31-16];
xmm1[31-16]     = (xmm1[31-16]      * xmm2/m128[31-16])[31-16];
xmm1[47-32]     = (xmm1[47-32]      * xmm2/m128[47-32])[31-16];
xmm1[63-48]     = (xmm1[63-48]      * xmm2/m128[63-48])[31-16];
xmm1[79-64]     = (xmm1[79-64]      * xmm2/m128[79-64])[31-16];
xmm1[95-80]     = (xmm1[95-80]      * xmm2/m128[95-80])[31-16];
xmm1[111-96]    = (xmm1[111-96]     * xmm2/m128[111-96])[31-16];
xmm1[127-112]   = (xmm1[127-112]    * xmm2/m128[127-112])[31-16];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.

#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PMULLW—Packed Multiply Low

Instruction	Description
PMULLW xmm1, xmm2/m128	Multiply the packed signed word integers in xmm1 by the packed signed word integers in xmm2/mem128, and store the low words of the results in xmm1.

Description

Multiplies the eight signed word integers in the destination operand by the corresponding eight signed word integers in the source operand, and store the low word of the individual doubleword results in the destination operand. The destination operand is an XMM register. The source operand can either be an XMM register or a 128-bit memory operand.

Operation

```
xmm1[15-0]      = (xmm1[15-0]      * xmm2/m128[15-0])[15-0];
xmm1[31-16]     = (xmm1[31-16]     * xmm2/m128[31-16])[15-0];
xmm1[47-32]     = (xmm1[47-32]     * xmm2/m128[47-32])[15-0];
xmm1[63-48]     = (xmm1[63-48]     * xmm2/m128[63-48])[15-0];
xmm1[79-64]     = (xmm1[79-64]     * xmm2/m128[79-64])[15-0];
xmm1[95-80]     = (xmm1[95-80]     * xmm2/m128[95-80])[15-0];
xmm1[111-96]    = (xmm1[111-96]    * xmm2/m128[111-96])[15-0];
xmm1[127-112]   = (xmm1[127-112]   * xmm2/m128[127-112])[15-0];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PMULUDQ—Multiply Doubleword Unsigned

Instruction	Description
PMULUDQ mm1, mm2/m64	Multiply unsigned doubleword integer in mm1 by unsigned doubleword integer in mm2/m64, and store the quadword result in mm1.

Description

Multiplies the low unsigned doubleword integer in the destination operand by an low unsigned doubleword integer in the source operand, and stores the quadword result in mm1. The destination operand is a MMX register. The source operand can either be a MMX register or a 64-bit memory operand.

Operation

```
mm1[63:0] = (mm1[31:0] * mm2/m64[31:0]);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

None.

PMULUDQ—Packed Multiply Doubleword Unsigned

Instruction	Description
PMULUDQ xmm1, xmm2/m128	Multiply packed unsigned doublewords in xmm1 by packed unsigned doublewords in xmm2/mem128, and then store the 64-bit results in xmm1.

Description

Multiplies two packed unsigned doublewords in the destination register by two packed unsigned doublewords in the source register and stores the packed quadword results in the destination register. The source operands used are the doublewords at bit positions 31-0 and 95-64. The quadword results are written to the destination operand. The destination operand is an XMM register. The source operand can either be an XMM register or a 128-bit memory operand.

Operation

```
xmm1[63:0] = (xmm1[31:0] * xmm2/m128[31:0]);
xmm1[127:64] = (xmm1[95:64] * xmm2/m128[95:64]);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.

If CR4.OSFXSR (bit 9) = 0.

If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

POR—Bitwise Logical OR

Instruction	Description
POR xmm1, xmm2/m128	Bitwise OR of xmm2/mem128 and xmm1.

Description

Performs a bitwise logical OR of the source operand (xmm/m128) and the destination operand (xmm1), and writes the result to the destination operand. Each bit of the result is set to 0 if the corresponding bits of the two operands are 0; otherwise, it is set to 1.

Operation

```
xmm1 |= xmm2/m128;
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PSADBW—Packed Sum of Absolute Differences

Instruction	Description
PSADBW xmm1,xmm2/m128	Absolute difference of packed unsigned byte integers from xmm2 /m128 and xmm1; these differences are then summed within separate high and low 64-bit sections to produce two word results.

Description

Computes the absolute value of the difference of unsigned bytes for xmm1 and xmm2/m128. These differences are then summed independently within the high and low 64-bit sections to produce two separate word results; the high 3 words of each 64-bit section are cleared. The destination operand is an XMM register. The source operand can either be an XMM register or a 128-bit memory operand.

Operation

```
    temp1    = ABS(xmm1[7-0]     - xmm2/m128[7-0]);
    temp2    = ABS(xmm1[15-8]    - xmm2/m128[15-8]);
    temp3    = ABS(xmm1[23-16]   - xmm2/m128[23-16]);
    temp4    = ABS(xmm1[31-24]   - xmm2/m128[31-24]);
    temp5    = ABS(xmm1[39-32]   - xmm2/m128[39-32]);
    temp6    = ABS(xmm1[47-40]   - xmm2/m128[47-40]);
    temp7    = ABS(xmm1[55-48]   - xmm2/m128[55-48]);
    temp8    = ABS(xmm1[63-56]   - xmm2/m128[63-56]);
    temp9    = ABS(xmm1[71-64]   - xmm2/m128[71-64]);
    temp10   = ABS(xmm1[79-72]   - xmm2/m128[79-72]);
    temp11   = ABS(xmm1[87-80]   - xmm2/m128[87-80]);
    temp12   = ABS(xmm1[95-88]   - xmm2/m128[95-88]);
    temp13   = ABS(xmm1[103-96]  - xmm2/m128[103-96]);
    temp14   = ABS(xmm1[111-104] - xmm2/m128[111-104]);
    temp15   = ABS(xmm1[119-112] - xmm2/m128[119-112]);
    temp16   = ABS(xmm1[127-120] - xmm2/m128[127-120]);

    xmm1[15:0] = temp1 + temp2 + temp3 + temp4 + temp5 + temp6 + temp7 + temp8;
    xmm1[31:16] = 0x00000000;
    xmm1[47:32] = 0x00000000;
    xmm1[63:48] = 0x00000000;
    xmm1[79:64] = temp9 + temp10 + temp11 + temp12 + temp13 + temp14 + temp15 +
temp16;
    xmm1[95:80] = 0x00000000;
    xmm1[111:96] = 0x00000000;
    xmm1[127:112] = 0x00000000;
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PSHUFD—Packed Shuffle Doubleword

Instruction	Description
PSHUFD xmm1, xmm2/m128, imm8	Shuffle the doublewords in xmm2/mem128 based on the encoding in imm8 and store result in xmm1.

Description

Shuffles the four doublewords in xmm2/mem128 in the order selected by imm8 and stores the result in xmm1. Bits 1 and 0 of imm8 encode the source for destination doubleword 0 (xmm1[31-0]), bits 3 and 2 encode for doubleword 1, bits 5 and 4 encode for doubleword 2, and bits 7 and 6 encode for doubleword 3 (xmm1[127-96]). Similarly, the two bit encoding represents which source doubleword is to be used, e.g., an binary encoding of 10 indicates that source doubleword 2 (xmm2/mem128[95-64]) will be used.

Operation

```
xmm1[31-0]   = (xmm2/m128 >> (imm8[1-0] * 32) )[31-0]
xmm1[63-32]  = (xmm2/m128 >> (imm8[3-2] * 32) )[31-0]
xmm1[95-64]  = (xmm2/m128 >> (imm8[5-4] * 32) )[31-0]
xmm1[127-96]  = (xmm2/m128 >> (imm8[7-6] * 32) )[31-0]
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.

#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PSHUFHW—Packed Shuffle High Words

Instruction	Description
PSHUFHW xmm1, xmm2/m128, imm8	Shuffle the high words in xmm2/mem128 based on the encoding in imm8 and store the result in xmm1.

Description

Shuffles the four high words in xmm2/mem128 in the order selected by imm8 and stores the result in the high quadword of xmm1. Bits 1 and 0 of imm8 encode the source for destination word 4 (xmm1[79-64]), bits 3 and 2 encode for word 5, bits 5 and 4 encode for word 6, and bits 7 and 6 encode for word 7 (xmm1[127-112]). Similarly, the two bit encoding represents which source word is to be used, e.g., a binary encoding of 10 indicates that source word 6 (XMM2[111-96] or Mem[111-96]) will be used. The low quadword of the destination register is written with the low 64 bits of the source register.

Operation

```
if (source == m128) {
    xmm1[79-64]   = (m128 >> (imm8[1-0] * 16) )[79-64]
    xmm1[95-80]   = (m128 >> (imm8[3-2] * 16) )[79-64]
    xmm1[111-96]  = (m128 >> (imm8[5-4] * 16) )[79-64]
    xmm1[127-112] = (m128 >> (imm8[7-6] * 16) )[79-64]

    xmm1[63-0] = m128[63-0];
} else {
    xmm1[79-64]   = (xmm2 >> (imm8[1-0] * 16) )[79-64]
    xmm1[95-80]   = (xmm2 >> (imm8[3-2] * 16) )[79-64]
    xmm1[111-96]  = (xmm2 >> (imm8[5-4] * 16) )[79-64]
    xmm1[127-112] = (xmm2 >> (imm8[7-6] * 16) )[79-64]

    xmm1[63-0] = xmm2[63-0];
}
```

Protected Mode Exceptions

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

	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PSHUFLW—Packed Shuffle Low Word

Instruction	Description
PSHUFLW xmm1, xmm2/m128, imm8	Shuffle the low words in xmm2/mem128 based on the encoding in imm8 and store in xmm1.

Description

Shuffles the four low words in xmm2/mem128 in the order selected by imm8 and stores the result in the low quadword of xmm1.Bits 1 and 0 of imm8 encode the source for destination word 0 (xmm1[15-0]), bits 3 and 2 encode for word 1, bits 5 and 4 encode for word 2, and bits 7 and 6 encode for word 3 (xmm1[63-48]). Similarly, the two bit encoding represents which source word is to be used, e.g., an binary encoding of 10 indicates that source word 2 (xmm2/mem128[47-32]) will be used. The high quadword of the destination register is written with the high 64 bits of the source register.

Operation

```
if (source == m128) {
    xmm1[15-0]   = (m128 >> (imm8[1-0] * 16) )[15-0]
    xmm1[31-16]  = (m128 >> (imm8[3-2] * 16) )[15-0]
    xmm1[47-32]  = (m128 >> (imm8[5-4] * 16) )[15-0]
    xmm1[63-48]  = (m128 >> (imm8[7-6] * 16) )[15-0]

    xmm1[127-64] = m128[127-64];
} else {
    xmm1[15-0]   = (xmm2 >> (imm8[1-0] * 16) )[15-0]
    xmm1[31-16]  = (xmm2 >> (imm8[3-2] * 16) )[15-0]
    xmm1[47-32]  = (xmm2 >> (imm8[5-4] * 16) )[15-0]
    xmm1[63-48]  = (xmm2 >> (imm8[7-6] * 16) )[15-0]

    xmm1[127-64] = xmm2[127-64];
}
```

Protected Mode Exceptions

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

	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PSLLDQ—Packed Shift Left Logical Double Quadword

Instruction	Description
PSLLDQ xmm1, imm8	Shift left xmm1 by imm8 bytes, clearing low-order bits.

Description

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

Operation

```
temp = imm8;
if (temp > 15) temp = 16;
xmm1 = xmm1 << (temp * 8);
```

Protected Mode Exceptions

#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PSLLW/PSLLD/PSLLQ — Packed Shift Left Logical

Instruction	Description
PSLLW xmm1, xmm2/m128	Shift words in xmm1 register left by amount specified in xmm2/mem128, while shifting in zeros.
PSLLW xmm1, imm8	Shift words in xmm1 left by imm8.
PSLLD xmm1, xmm2/m128	Shift doublewords in xmm1 left by amount specified in xmm2/mem128, while shifting in zeros.
PSLLD xmm1, imm8	Shift doublewords in xmm1 by imm8.
PSLLQ xmm1, xmm2/m128	Shift quadwords in xmm1 left by amount specified in xmm2/mem128, while shifting in zeros.
PSLLQ xmm1, imm8	Shift quadwords in xmm1 by imm8.

Description

Shift the bits of the first operand to the left by the number of bits specified in the count operand. The result of the shift operation is written to the destination register. The empty low-order bits are cleared (set to zero). If the value specified by the second operand is greater than 15 (for words), 31 (for doublewords), or 63 (for quadwords) then the destination is set to all zeros. If the shift amount is specified by xmm2/m128, only the low 64 bits of this operand are used; the high 64 bits are ignored. The destination operand is an XMM register. The count operand (second operand) can be either an XMM register, a 128-bit memory operand, or an immediate 8-bit operand.

The PSLLW instruction shifts each of the eight words in the destination register to the left by the number of bits specified in the count operand. The low-order bit positions (of each word) are filled with zeros.

The PSLLD instruction shifts each of the four doublewords in the destination register to the left by the number of bits specified in the count operand. The empty low-order bit positions (of each doubleword) are filled with zeros.

The PSLLQ instruction shifts each of the two quadwords in the destination register to the left by the number of bits specified in the count operand. The empty low-order bit positions (of each quadword) are filled with zeros.

Operation

```
if (second_operand == imm8) {
    temp = imm8;
}
else {
    // second operand is xmm2/m128
    temp = xmm2/m128[63-0];
}
if (instruction == PSLLW) {
    xmm1[15-0]   = xmm1[15-0]   << temp;
    xmm1[31-16]  = xmm1[31-16]  << temp;
    xmm1[47-32]  = xmm1[47-32]  << temp;
    xmm1[63-48]   = xmm1[63-48]   << temp;
    xmm1[79-64]   = xmm1[79-64]   << temp;
    xmm1[95-80]   = xmm1[95-80]   << temp;
    xmm1[111-96]  = xmm1[111-96]  << temp;
    xmm1[127-112] = xmm1[127-112] << temp;
}
else if (instruction == PSLLD) {
    // instruction is PSLLD
    xmm1[31-0]    = xmm1[31-0]    << temp;
    xmm1[63-32]   = xmm1[63-32]   << temp;
    xmm1[95-64]   = xmm1[95-64]   << temp;
    xmm1[127-96] = xmm1[127-96] << temp;
}
else {
    // instruction is PSLLQ
    xmm1[63-0]    = xmm1[63-0]    << temp;
    xmm1[127-64]          = xmm1[127-64]          << temp;
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.

If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PSRAW/PSRAD—Packed Shift Right Arithmetic

Instruction	Description
PSRAW xmm1, xmm2/m128	Shift words in xmm1 right by amount specified in xmm2/mem128 while shifting in sign bits.
PSRAW xmm1, imm8	Shift words in xmm1 right by imm8 while shifting in sign bits
PSRAD xmm1, xmm2/m128	Shift doubleword in xmm1 right by amount specified in xmm2 /m128 while shifting in sign bits.
PSRAD xmm1, imm8	Shift doublewords in xmm1 right by imm8 while shifting in sign bits.

Description

Shift the bits of the first operand to the right by the number of bits specified in the count operand. The result of the shift operation is written to the destination register. The empty high-order bits of each element are filled with the initial value of the sign bit of the data element. The destination operand is an XMM register. The count operand (second operand) can be either by an XMM register, a 128-bit memory operand, or an immediate 8-bit operand.

The PSRAW instruction shifts each of the eight words in the destination register to the right by the number of bits specified in the count operand. If the value specified by the second operand is greater than 15, each destination element is filled with the initial value of the sign bit of the element.

The PSRAD instruction shifts each of the four doublewords in the destination register to the right by the number of bits specified in the count operand.If the value specified by the second operand is greater than 31, each destination element is filled with the initial value of the sign bit of the element.

Operation

```
if (second_operand == imm8) {
    temp = imm8;
}
else {
    // second operand is xmm2/m128
    temp = xmm2/m128;
}
if (instruction == PSRAW) {
    xmm1[15-0]    = SignExtend  (xmm1[15-0]    >> temp);
    xmm1[31-16]   = SignExtend  (xmm1[31-16]   >> temp);
    xmm1[47-32]   = SignExtend  (xmm1[47-32]   >> temp);
    xmm1[63-48]   = SignExtend  (xmm1[63-48]   >> temp);
    xmm1[79-64]   = SignExtend  (xmm1[79-64]   >> temp);
    xmm1[95-80]   = SignExtend  (xmm1[95-80]   >> temp);
    xmm1[111-96]  = SignExtend  (xmm1[111-96]  >> temp);
    xmm1[127-112] = SignExtend (xmm1[127-112] >> temp);
}
else {
    // instruction is PSRAD
    xmm1[31-0]   = SignExtend (xmm1[31-0]   >> temp);
    xmm1[63-32] = SignExtend  (xmm1[63-32]  >> temp);
    xmm1[95-64]  = SignExtend (xmm1[95-64]  >> temp);
    xmm1[127-96] = SignExtend (xmm1[127-96] >> temp);
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PSRLDQ—Packed Shift Right Logical Double Quadword

Instruction	Description
PSRLDQ xmm1, imm8	Shift right xmm1 by imm8, clearing high-order bits.

Description

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

Operation

```
temp = imm8;
if (temp > 15) temp = 16;
xmm1 = xmm1 >> (temp * 8);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.

If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PSRLW/PSRLD/PSRLQ— Packed Shift Right Logical

Instruction	Description
PSRLW xmm1, xmm2/m128	Shift words in xmm1 right by amount specified in xmm2/mem128 while shifting in zeroes.
PSRLW xmm1, imm8	Shift words in xmm1 right by imm8.
PSRLD xmm1, xmm2/m128	Shift doublewords in xmm1 right by amount specified in XMM2 /Mem while shifting in zeroes.
PSRLD xmm1, imm8	Shift doublewords in xmm1 right by imm8.
PSRLQ xmm1, xmm2/m128	Shift quadwords in xmm1 right by amount specified in xmm2/mem128 while shifting in zeroes.
PSRLQ xmm1, imm8	Shift quadwords in xmm1 right by imm8.

Description

Shift the bits of the first operand to the right by the number of bits specified in the count operand. The result of the shift operation is written to the destination register. The empty high-order bits are cleared (set to zero). If the value specified by the second operand is greater than 15 (for words), 31 (for doublewords), or 63 (for quadwords) then the destination is set to all zeros. If the shift amount is specified by xmm2/m128, only the low 64 bits of this operand are used; the high 64 bits are ignored. The destination operand is an XMM register. The count operand (second operand) can be either an XMM register, a 128-bit memory operand, or an immediate 8-bit operand.

The PSRLW instruction shifts each of the eight words in the destination register to the right by the number of bits specified in the count operand. The empty high-order bits (of each word) are filled with zeros.

The PSRLD instruction shifts each of the four doublewords in the destination register to the right by the number of bits specified in the count operand. The empty high-order bits (of each double-word) are filled with zeros.

The PSRLQ instruction shifts each of the two quadwords in the destination register to the right by the number of bits specified in the count operand. The empty high-order bits (of each quad-word) are filled with zeros.

Operation

```
if (second_operand == imm8) {
    temp = imm8;
}
else {
    // second operand is xmm2/m128
    temp = xmm2/m128[63-0];
}

    if (instruction == PSRLW) {
    xmm1[15-0]    = xmm1[15-0]     >> temp;
    xmm1[31-16]   = xmm1[31-16]    >> temp;
    xmm1[47-32]   = xmm1[47-32]    >> temp;
    xmm1[63-48]   = xmm1[63-48]    >> temp;
    xmm1[79-64]   = xmm1[79-64]    >> temp;
    xmm1[95-80]   = xmm1[95-80]    >> temp;
    xmm1[111-96]   = xmm1[111-96]    >> temp;
    xmm1[127-112]       = xmm1[127-112]            >> temp;
    }
    else if (instruction == PSRLD) {
    // instruction is PSRLD
    xmm1[31-0]   = xmm1[31-0]   >> temp;
    xmm1[63-32]  = xmm1[63-32]  >> temp;
    xmm1[95-64]  = xmm1[95-64]  >> temp;
    xmm1[127-96] = xmm1[127-96] >> temp;
}
else {
    // instruction is PSRLQ
    xmm1[63-0]   = xmm1[63-0]   >> temp;
    xmm1[127-64] = xmm1[127-64] >> temp;
                        }
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.

	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PSUBB/PSUBW/PSUBD—Packed Subtract

Instruction	Description
PSUBB xmm1, xmm2/m128	Subtract packed bytes in xmm2/mem128 from packed bytes in xmm1.
PSUBW xmm1, xmm2/m128	Subtract packed words in xmm2/mem128 from packed words in xmm1.
PSUBD xmm1, xmm2/m128	Subtract packed doublewords in xmm2 /mem128 from packed doublewords in xmm1.

Description

Subtracts the data elements of the source operand from the corresponding data elements of the destination register and writes the results to the destination register. If the result is larger or smaller than the architectural limit, it wraps around. The destination operand is an XMM register. The source operand can either be an XMM register or a 128-bit memory operand.

The PSUBB instruction subtracts the bytes of the source operand xmm2/m128 from the bytes of the destination operand xmm1.When a result is too large or too small to be represented in a byte, the result wraps around and the low 8 bits are written to the destination element.

The PSUBW instruction subtracts the words of the source operand xmm2/m128 from the words of the destination operand xmm1.When a result is too large or too small to be represented in a word, the result wraps around and the low 16 bits are written to the destination element.

The PSUBD instruction subtracts the doublewords of the source operand xmm2/m128 from the doublewords of the destination operand xmm1. When a result is too large or too small to be represented in a doubleword, the result wraps around and the low 32 bits are written to the destination element.

Operation

```
if (instruction == PSUBB){
    xmm1[7-0]     = xmm1[7-0]     - xmm2/m128[7-0];
    xmm1[15-8]    = xmm1[15-8]    - xmm2/m128[15-8];
    xmm1[23-16]   = xmm1[23-16]   - xmm2/m128[23-16];
    xmm1[31-24]   = xmm1[31-24]   - xmm2/m128[31-24];
    xmm1[39-32]   = xmm1[39-32]   - xmm2/m128[39-32];
    xmm1[47-40]   = xmm1[47-40]   - xmm2/m128[47-40];
    xmm1[55-48]   = xmm1[55-48]   - xmm2/m128[55-48];
    xmm1[63-56]   = xmm1[63-56]   - xmm2/m128[63-56];
    xmm1[71-64]   = xmm1[71-64]   - xmm2/m128[71-64];
    xmm1[79-72]   = xmm1[79-72]   - xmm2/m128[79-72];
    xmm1[87-80]   = xmm1[87-80]   - xmm2/m128[87-80];
    xmm1[95-88]   = xmm1[95-88]   - xmm2/m128[95-88];
    xmm1[103-96]  = xmm1[103-96]  - xmm2/m128[103-96];
    xmm1[111-104] = xmm1[111-104] - xmm2/m128[111-104];
    xmm1[119-112] = xmm1[119-112] - xmm2/m128[119-112];
    xmm1[127-120] = xmm1[127-120] - xmm2/m128[127-120];
}
```

```
else if (instruction == PSUBW){
    xmm1[15-0]      = xmm1[15-0]     - xmm2/m128[15-0];
    xmm1[31-16]     = xmm1[31-16]    - xmm2/m128[31-16];
    xmm1[47-32]     = xmm1[47-32]    - xmm2/m128[47-32];
    xmm1[63-48]     = xmm1[63-48]    - xmm2/m128[63-48];
    xmm1[79-64]     = xmm1[79-64]    - xmm2/m128[79-64];
    xmm1[95-80]     = xmm1[95-80]    - xmm2/m128[95-80];
    xmm1[111-96]    = xmm1[111-96]   - xmm2/m128[111-96];
    xmm1[127-112]   = xmm1[127-112]  - xmm2/m128[127-112];
}
else {
    // instruction is PSUBD
    xmm1[31-0]      = xmm1[31-0]     - xmm2/m128[31-0];
    xmm1[63-32]     = xmm1[63-32]    - xmm2/m128[63-32];
    xmm1[95-64]     = xmm1[95-64]    - xmm2/m128[95-64];
    xmm1[127-96]    = xmm1[127-96]   - xmm2/m128[127-96];
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PSUBQ—Packed Subtract Quadword

Instruction	Description
PSUBQ mm1,mm2/m64	Subtract quadword from mm2 /m64 from quadword in mm1.

Description

Subtracts the quadword from the source operand mm2/m64 from the quadword in the destination operand mm1 and writes the result to the mm1 register. When the result is too large to be represented in a packed quadword (overflow), the result wraps around and the low 64 bits are written to the destination register (that is, the carry is ignored). The destination operand is a MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

Note that like the integer SUB instruction, the PSUBQ instruction can operate on either unsigned or signed (two's complement notation) integers. Unlike the integer instructions, none of the 64-bit or 128-bit integer instructions affect the EFLAGS register. With these integer instructions, there are no carry or overflow flags to indicate when overflow has occurred, so the software must control the range of values.

Operation

```
mm1[63-0] = mm1[63-0] - mm2/m64[63-0];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.

	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference if the current privilege level is 3.

Numeric Exceptions

None.

PSUBQ—Packed Subtract Quadword

Instruction	Description
PSUBQ xmm1,xmm2/m128	Subtract quadwords from xmm2 /m128 from quadwords in xmm1.

Description

Subtracts the packed quadwords from the source operand xmm2/m128 from the corresponding quadwords in the destination operand xmm1 and writes the results to the xmm1 register. When an individual result is too large to be represented in a packed quadword (overflow), the result wraps around and the low 64 bits are written to the destination register (that is, the carry is ignored). The destination operand is an XMM register. The source operand can either be an XMM register or a 128-bit memory operand.

Note that like the integer SUB instruction, the PSUBQ instruction can operate on either unsigned or signed (two’s complement notation) integers. Unlike the integer instructions, none of the 64-bit or 128-bit integer instructions affect the EFLAGS register. With these integer instructions, there are no carry or overflow flags to indicate when overflow has occurred, so the software must control the range of values.

Operation

```
xmm1[63-0] = xmm1[63-0] - xmm2/m128[63-0];
xmm1[127-64] = xmm1[127-64] - xmm2/m128[127-64];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PSUBSB/PSUBSW—Packed Subtract with Saturation

Instruction	Description
PSUBSB xmm1, xmm2/m128	Subtract packed signed byte integers in xmm2/mem128 from packed signed byte integers in xmm1 and saturate.
PSUBSW xmm1, xmm2/m128	Subtract packed signed word integers in xmm2/mem128 from packed signed word integers in xmm1 and saturate.

Description

Subtracts the data elements of the source operand from the corresponding data elements of the destination operand. The result is saturated to the limits of a signed data element and is written to the destination operand. The destination operand is an XMM register. The source operand can either be a 128-bit XMM register or a 128-bit memory operand.

The PSUBB instruction subtracts the signed byte integers of the source operand xmm2/m128 from the signed byte integers of the destination operand xmm1, and writes the results to the destination register xmm1. If an individual is larger or smaller than the range of a signed byte, the value is saturated; in the case of an overflow to 7FH and in the case of an underflow to 80H.

The PSUBW instruction subtracts the signed word integers of the source operand xmm2/m128 from the signed word integers of the destination operand xmm1 and writes the results to the destination register xmm1. If an individual result is larger or smaller than the range of a signed word, the value is saturated; in the case of an overflow to 7FFFH, and in the case of an underflow to 8000H.

Operation

```
if (instruction == PSUBSB) {
    xmm1[7-0]     = SaturateToSignedByte (xmm1[7-0]     - xmm2/m128[7-0]);
    xmm1[15-8]    = SaturateToSignedByte (xmm1[15-8]    - xmm2/m128[15-8]);
    xmm1[23-16]   = SaturateToSignedByte (xmm1[23-16]   - xmm2/m128[23-16]);
    xmm1[31-24]   = SaturateToSignedByte (xmm1[31-24]   - xmm2/m128[31-24]);
    xmm1[39-32]   = SaturateToSignedByte (xmm1[39-32]   - xmm2/m128[39-32]);
    xmm1[47-40]   = SaturateToSignedByte (xmm1[47-40]   - xmm2/m128[47-40]);
    xmm1[55-48]   = SaturateToSignedByte (xmm1[55-48]   - xmm2/m128[55-48]);
    xmm1[63-56]   = SaturateToSignedByte (xmm1[63-56]   - xmm2/m128[63-56]);
    xmm1[71-64]   = SaturateToSignedByte (xmm1[71-64]   - xmm2/m128[71-64]);
    xmm1[79-72]   = SaturateToSignedByte (xmm1[79-72]   - xmm2/m128[79-72]);
    xmm1[87-80]   = SaturateToSignedByte (xmm1[87-80]   - xmm2/m128[87-80]);
    xmm1[95-88]   = SaturateToSignedByte (xmm1[95-88]   - xmm2/m128[95-88]);
    xmm1[103-96]  = SaturateToSignedByte (xmm1[103-96]  - xmm2/m128[103-96]);
    xmm1[111-104] = SaturateToSignedByte (xmm1[111-104] - xmm2/m128[111-104]);
    xmm1[119-112] = SaturateToSignedByte (xmm1[119-112] - xmm2/m128[119-112]);
    xmm1[127-120] = SaturateToSignedByte (xmm1[127-120] - xmm2/m128[127-120]);
}
else {
    // instruction is PSUBW
    xmm1[15-0]    = SaturateToSignedWord (xmm1[15-0]    - xmm2/m128[15-0]);
    xmm1[31-16]   = SaturateToSignedWord (xmm1[31-16]   - xmm2/m128[31-16]);
    xmm1[47-32]   = SaturateToSignedWord (xmm1[47-32]   - xmm2/m128[47-32]);
    xmm1[63-48]   = SaturateToSignedWord (xmm1[63-48]   - xmm2/128[63-48]);
    xmm1[79-64]   = SaturateToSignedWord (xmm1[79-64]   - xmm2/m128[79-64]);
    xmm1[95-80]   = SaturateToSignedWord (xmm1[95-80]   - xmm2/m128[95-80]);
    xmm1[111-96]  = SaturateToSignedWord (xmm1[111-96]  - xmm2/m128[111-96]);
    xmm1[127-112] = SaturateToSignedWord (xmm1[127-112] - xmm2/m128[127-112]);
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PSUBUSB/PSUBUSW—Packed Subtract Unsigned with Saturation

Instruction	Description
PSUBUSB xmm1, xmm2/m128	Subtract packed unsigned byte integers in xmm2/mem128 from packed unsigned byte integers in xmm1 and saturate.
PSUBUSW xmm1, xmm2/m128	Subtract packed unsigned word integers in xmm2/mem128 from packed unsigned word integers in xmm1 and saturate.

Description

Subtract the data elements of the source operand from the data elements of the destination register. The results are saturated to the limits of an unsigned data element and written to the destination operand. The destination operand is an XMM register. The source operand can either be an XMM register or a 128-bit memory operand.

The PSUBUSB instruction subtracts the unsigned byte integers of the source operand xmm2/m128 from the unsigned byte integers of the destination operand xmm1 and writes the results to the destination register xmm1. In the case where a result element is less than zero (a negative value), it is saturated to 00H.

The PSUBUSW instruction subtracts the unsigned word integers of the source operand xmm2/m128 from the unsigned word integers of the destination operand xmm1 and writes the results to the destination register xmm1. In the case where a result element is less than zero (a negative value), it is saturated to 0000H.

Operation

```
if (instruction == PSUBUSB) {
    xmm1[7-0]     = SaturateToUnsignedByte (xmm1[7-0]     - xmm2/m128[7-0]);
    xmm1[15-8]    = SaturateToUnsignedByte (xmm1[15-8]    - xmm2/m128[15-8]);
    xmm1[23-16]   = SaturateToUnsignedByte (xmm1[23-16]   - xmm2/m128[23-16]);
    xmm1[31-24]   = SaturateToUnsignedByte (xmm1[31-24]   - xmm2/m128[31-24]);
    xmm1[39-32]   = SaturateToUnsignedByte (xmm1[39-32]   - xmm2/m128[39-32]);
    xmm1[47-40]   = SaturateToUnsignedByte (xmm1[47-40]   - xmm2/m128[47-40]);
    xmm1[55-48]   = SaturateToUnsignedByte (xmm1[55-48]   - xmm2/m128[55-48]);
    xmm1[63-56]   = SaturateToUnsignedByte (xmm1[63-56]   - xmm2/m128[63-56]);
    xmm1[79-64]   = SaturateToUnSignedByte (xmm1[79-64]   - xmm2/m128[79-64]);
    xmm1[95-80]   = SaturateToUnSignedByte (xmm1[95-80]   - xmm2/m128[95-80]);
    xmm1[111-96]  = SaturateToUnSignedByte (xmm1[111-96]  - xmm2/m128[111-96]);
    xmm1[127-112] = SaturateToUnSignedByte (xmm1[127-112] - xmm2/m128[127-112]);
}
else {
    // instruction is PSUBUSW
    xmm1[15-0]    = SaturateToUnsignedWord (xmm1[15-0]    - xmm2/m128[15-0]);
    xmm1[31-16]   = SaturateToUnsignedWord (xmm1[31-16]   - xmm2/m128[31-16]);
    xmm1[47-32]   = SaturateToUnsignedWord (xmm1[47-32]   - xmm2/m128[47-32]);
    xmm1[63-48]   = SaturateToUnsignedWord (xmm1[63-48]   - xmm2/m128[63-48]);
    xmm1[79-64]   = SaturateToUnSignedWord (xmm1[79-64]   - xmm2/m128[79-64]);
    xmm1[95-80]   = SaturateToUnSignedWord (xmm1[95-80]   - xmm2/m128[95-80]);
    xmm1[111-96]  = SaturateToUnSignedWord (xmm1[111-96]  - xmm2/m128[111-96]);
    xmm1[127-112] = SaturateToUnSignedWord (xmm1[127-112] - xmm2/m128[127-112]);
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

PUNPCKH—Unpack High Packed Data

Instruction	Description
PUNPCKHBW xmm1, xmm2/m128	Interleave bytes from the high quadwords of xmm1 and xmm2/mem128 into xmm1.
PUNPCKHWD xmm1, xmm2/m128	Interleave words from the high quadwords of xmm1 and xmm2/mem128 into xmm1.
PUNPCKHDQ xmm1, xmm2/m128	Interleave doublewords from the high quadwords of xmm1 and xmm2/mem128 into xmm1.
PUNPCKHQDQ xmm1, xmm2/m128	Interleave high quadwords of xmm1 and xmm2/mem128 into xmm1

Description

Unpack and interleave the high-order data elements of the destination and source operands into the destination operand. The low-order data elements are ignored. The destination operand is an XMM register. The source operand can either be an XMM register or a 128-bit memory operand.

The PUNPCKHBW instruction interleaves the eight high-order bytes of the source operand and the eight high-order bytes of the destination operand and writes them to the destination register.

The PUNPCKHWD instruction interleaves the four high-order words of the source operand and the four high-order words of the destination operand and writes them to the destination register.

The PUNPCKHDQ instruction interleaves the two high-order doublewords of the source operand and the two high-order doublewords of the destination operand and writes them to the destination register.

The PUNPCKHQDQ instruction interleaves the high-order quadword of the source operand and the high-order quadword of the destination operand and writes them to the destination register.

Operation

```
if (instruction == PUNPCKHBW) {
    xmm1[7-0]     = xmm1[71-64];
    xmm1[15-8]    = xmm2/m128[71-64];
    xmm1[23-16]   = xmm1[79-72];
    xmm1[31-24]   = xmm2/m128[79-72];
    xmm1[39-32]   = xmm1[87-80];
    xmm1[47-40]   = xmm2/m128[87-80];
    xmm1[55-48]   = xmm1[95-88];
    xmm1[63-56]   = xmm2/m128[95-88];
    xmm1[71-64]   = xmm1[103-96];
    xmm1[79-72]   = xmm2/m128[103-96];
    xmm1[87-80]   = xmm1[111-104];
    xmm1[95-88]   = xmm2/m128[111-104];
    xmm1[103-96]  = xmm1[119-112];
    xmm1[111-104] = xmm2/m128[119-112];
    xmm1[119-112] = xmm1[127-120];
    xmm1[127-120] = xmm2/m128[127-120];
}
else if (instruction == PUNPCKHWD) {
    xmm1[15-0]    = xmm1[79-64];
    xmm1[31-16]   = xmm2/m128[79-64];
    xmm1[47-32]   = xmm1[95-80];
    xmm1[63-48]   = xmm2/m128[95-80];
    xmm1[79-64]   = xmm1[111-96];
    xmm1[95-80]   = xmm2/m128[111-96];
    xmm1[111-96]  = xmm1[127-112];
    xmm1[127-112] = xmm2/m128[127-112];
}
else if (instruction == PUNPCKHDQ) {
    xmm1[31-0]   = xmm1[95-64];
    xmm1[63-32]  = xmm2/m128[95-64];
    xmm1[95-64]  = xmm1[127-96];
    xmm1[127-96] = xmm2/m128[127-96];
}
else if (instruction == PUNPCKHQDQ) {
    xmm1[63-0]   = xmm1[127-64];
    xmm1[127-64] = xmm2/m128[127-64];
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.

#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

Comments

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits. Alignment to 16-byte boundary and normal segment checking will still be enforced.

PUNPCKL—Unpack Low Packed Data

Instruction	Description
PUNPCKLBW xmm1, xmm2/m128	Interleave bytes from the low quadwords of xmm1 and xmm2/mem128 into xmm1.
PUNPCKLWD xmm1, xmm2/m128	Interleave words from the low quadwords of xmm1 and xmm2/mem128 into xmm1.
PUNPCKLDQ xmm1, xmm2/m128	Interleave doublewords from the low quadwords of xmm1 and xmm2/mem128 into xmm1.
PUNPCKLQDQ xmm1, xmm2/m128	Interleave the low quadwords of xmm1 and xmm2/mem128 into xmm1 register

Description

Unpack and interleave the low-order data elements of the destination and source operands into the destination operand. The high-order data elements are ignored. The destination operand is an XMM register. The source operand can either be an XMM register or a 128-bit memory operand.

When unpacking from a memory operand, the full 128-bit operand is accessed from memory. The instruction uses only the low-order 64 bits.

The PUNPCKLBW instruction interleaves the eight low-order bytes of the source operand and the eight low-order bytes of the destination operand and writes them to the destination register.

The PUNPCKLWD instruction interleaves the four low-order words of the source operand and the four low-order words of the destination operand and writes them to the destination register.

The PUNPCKLDQ instruction interleaves the two low-order doublewords of the source operand and the two low-order doublewords of the destination operand and writes them to the destination register.

The PUNPCKLQDQ instruction interleaves the low-order quadword of the source operand and the low-order quadword of the destination operand and writes them to the destination register.

Operation

```
if (instruction == PUNPCKLBW) {
    xmm1[7-0]     = xmm1[7-0];
    xmm1[15-8]    = xmm2/m128[7-0];
    xmm1[23-16]   = xmm1[15-8];
    xmm1[31-24]   = xmm2/m128[15-8];
    xmm1[39-32]   = xmm1[23-16];
    xmm1[47-40]   = xmm2/m128[23-16];
    xmm1[55-48]   = xmm1[31-24];
    xmm1[63-56]   = xmm2/m128[31-24];
    xmm1[71-64]   = xmm1[39-32];
    xmm1[79-72]   = xmm2/m128[39-32];
    xmm1[87-80]   = xmm1[47-40];
    xmm1[95-88]   = xmm2/m128[47-40];
    xmm1[103-96]  = xmm1[55-48];
    xmm1[111-104] = xmm2/m128[55-48];
    xmm1[119-112] = xmm1[63-56];
    xmm1[127-120] = xmm2/m128[63-56];
}
else if (instruction == PUNPCKLWD) {
    xmm1[15-0]    = xmm1[15-0];
    xmm1[31-16]   = xmm2/m128[15-0];
    xmm1[47-32]   = xmm1[31-16];
    xmm1[63-48]   = xmm2/m128[31-16];
    xmm1[79-64]   = xmm1[47-32];
    xmm1[95-80]   = xmm2/m128[47-32];
    xmm1[111-96]  = xmm1[63-48];
    xmm1[127-112] = xmm2/m128[63-48];
}
else if (instruction == PUNPCKLDQ) {
    xmm1[31-0]   = xmm1[31-0];
    xmm1[63-32]  = xmm2/m128[31-0];
    xmm1[95-64]  = xmm1[63-32];
    xmm1[127-96] = xmm2/m128[63-32];
}
else if (instruction == PUNPCKLQDQ) {
    xmm1[63-0]   = xmm1[63-0];
    xmm1[127-64] = xmm2/m128[63-0];
}
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

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

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

Comments

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits. Alignment to 16-byte boundary and normal segment checking will still be enforced.

PXOR—Bitwise Logical XOR

Instruction	Description
PXOR xmm1, xmm2/m128	Bitwise XOR of xmm2/mem128 and xmm1.

Description

Performs a bitwise logical XOR of the destination and source operands and writes the result to the destination register. Each bit of the result is 1 if the corresponding bits of the two operands are different. Each bit is 0 if the corresponding bits of the operands are the same. The source operand can either be an XMM register or a 128-bit memory operand.

Operation

```
xmm1 ^= xmm2/m128;
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

3.4. CACHEABILITY CONTROL AND MEMORY ORDERING INSTRUCTIONS

This chapter describes the cacheability control instructions, which enable programs to minimize data access latency and cache pollution, and the memory ordering instructions.

CLFLUSH—Cache Line Flush

Instruction	Description
CLFLUSH m8	Cache line containing m8 is flushed and invalidated from all caches in the coherency domain.

Description

Invalidates the cache line associated with the linear address specified by the value of m8 from all levels of the processor cache hierarchy (data and instruction). The invalidation is broadcast throughout the coherence domain. If, at any level of the cache hierarchy, the line is inconsistent with memory (dirty) it is written to memory before invalidation.

The line size affected is at least 64-bytes (aligned on a 64-byte boundary) and the region flushed must contain the m8 address. An implementation may flush a larger region.

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction.

CLFLUSH is an unordered operation with respect to other memory traffic including other CLFLUSH instructions. Software should use an SFENCE (memory fence) or a serializing instruction (as needed) before or after using CLFLUSH for cases where ordering is a concern. For example, software can use an SFENCE instruction to insure that previous stores are included in the write-back.

The CLFLUSH instruction can be used at all privileged levels and is subject to all permission checking and faults associated with a byte load. Like a load, the CLFLUSH instruction sets the A bit but not the D bit in the page tables.

Operation

Cache line containing m8 is flushed and invalidated from all caches in the coherency domain.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#UD	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

LFENCE—Load Fence

Instruction	Description
LFENCE	Guarantees that every load instruction that precedes, in program order, the load fence instruction is globally visible before any load instruction which follows the fence in program order is globally visible.

Description

Guarantees ordering between two loads and prevents speculative loads from passing the LFENCE instruction. LFENCE is ordered with respect to other LFENCE, MFENCE instructions and serializing instructions (such as CPUID.). It is not ordered with respect to stores or the SFENCE instruction.

Operation

```
while (!(preceding_loads_globally_visible)) wait();
```

Protected Mode Exceptions

None.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

None.

Numeric Exceptions

None.

Comments

LFENCE ignores the value of CR4.OSFXSR. LFENCE will not generate an invalid exception if CR4.OSFXSR = 0

MASKMOVDQU—Byte Mask Write Unaligned

Instruction	Description
MASKMOVDQU xmm1, xmm2	Move 128 bits representing integer data from xmm1 to memory location specified by the EDI register, using the byte mask in xmm2.

Description

Stores the contents of the xmm1 register to the location specified by the DI/EDI register (using DS segment). The size of the store address depends on the address-size attribute. The most significant bit in each byte of the mask register mm2 is used to selectively write the data (0 = no write, 1 = write), on a per-byte basis. Behavior with a mask of all zeroes is as follows:

- No data will be written to memory.
- For memory references, a zero byte mask does not prevent addressing faults (i.e., #GP, #SS) from being signaled.
- Signaling of page faults (#PF) is implementation-specific.
- The #UD, #NM, #MF, and #AC exceptions are signaled irrespective of the value of the mask.
- Signaling of breakpoints (code or data) is not guaranteed; different processor implementations may signal or not signal these breakpoints.
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (i.e., is reserved) and is implementation-specific. Dependence on the behavior of a specific implementation in this case is not recommended, and may lead to future incompatibility.

Operation

```
if (xmm2[7])   m128[edi]    = xmm1[7-0];
if (xmm2[15])  m128[edi+1]  = xmm1[15-8];
if (xmm2[23])  m128[edi+2]  = xmm1[23-16];
if (xmm2[31])  m128[edi+3]  = xmm1[31-24];
if (xmm2[39])  m128[edi+4]  = xmm1[39-32];
if (xmm2[47])  m128[edi+5]  = xmm1[47-40];
if (xmm2[55])  m128[edi+6]  = xmm1[55-48];
if (xmm2[63])  m128[edi+7]  = xmm1[63-56];
if (xmm2[71])  m128[edi+8]  = xmm1[71-64];
if (xmm2[79])  m128[edi+9]  = xmm1[79-72];
if (xmm2[87])  m128[edi+10]  = xmm1[87-80];
if (xmm2[95])  m128[edi+11]  = xmm1[95-88];
if (xmm2[103])  m128[edi+12]  = xmm1[103-96];
if (xmm2[111])  m128[edi+13]  = xmm1[111-104];
if (xmm2[119])  m128[edi+14]  = xmm1[119-112];
if (xmm2[127])  m128[edi+15]  = xmm1[127-120];
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

Comments

MASKMOVDQU can be used to improve performance for algorithms which need to merge data on a byte granularity. MASKMOVDQU should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store. Similar to the Streaming SIMD Extensions non-temporal store instructions, MASKMOVDQU minimizes pollution of the cache hierarchy. MASKMOVDQU implicitly uses weakly-ordered, write-combining stores (WC). See Section 2.4.3.2., “Caching of Temporal Vs. Non-Temporal Data” for further information about non-temporal stores.

As a consequence of the resulting weakly-ordered memory consistency model, a fencing operation such as SFENCE should be used if multiple processors may use different memory types to read/write the same memory location specified by EDI.

MFENCE—Memory Fence

Instruction	Description
MFENCE	Guarantees that every memory access that precedes, in program order, the memory fence instruction is globally visible before any memory instruction that follows the fence, in program order, is globally visible.

Description

Ensures that all prior loads and stores before the memory fence are globally observed prior to any loads or stores after the memory fence. MFENCE is ordered with respect to other explicit fences (SFENCE, LFENCE, and other MFENCE instructions) and serializing instructions (such as CPUID).

Operation

```
while (!(preceding_loads&stores_globally_visible)) wait();
```

Protected Mode Exceptions

None.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

None.

Numeric Exceptions

None.

Comments

MFENCE ignores the value of CR4.OSFXSR. MFENCE will not generate an invalid exception if CR4.OSFXSR = 0

MOVNTPD—Move Aligned Four Packed Double-Precision Floating-Point Non-Temporal

Instruction	Description
MOVNTPD m128, xmm	Move 128 bits representing packed double-precision floating-point data from xmm to m128, minimizing pollution in the cache hierarchy.

Description

Moves the double quadword representing packed double-precision floating-point data from XMM register to m128, minimizing pollution in the cache hierarchy. The linear address corresponds to the m128 address of the least-significant byte of the referenced memory data. This store instruction minimizes cache pollution.

Operation

```
m128 = xmm;
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.

If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

Comments

MOVTNPD should be used when dealing with 16-byte aligned double-precision floating-point numbers. MOVNTPD minimizes pollution in the cache hierarchy. As a consequence of the resulting weakly-ordered memory consistency model, a fencing operation such as SFENCE should be used if multiple processors might use different memory types to read/write the memory location. See Section 2.4.3.2., "Caching of Temporal Vs. Non-Temporal Data" for further information about non-temporal stores.

MOVNTDQ—Move Double Quadword Non-Temporal

Instruction	Description
MOVNTDQ m128, xmm	Move double quadword representing integer operands (bytes, words, doublewords, quadwords) from xmm to m128, minimizing pollution within cache hierarchy.

Description

Moves the double quadword representing integer operands (bytes, words, doublewords, quadwords) from XMM register to m128, minimizing pollution within cache hierarchy The linear address corresponds to the address of the least-significant byte of the referenced memory data. This store instruction minimizes cache pollution.

Operation

```
m128 = xmm;
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

Comments

MOVNTDQ minimizes pollution in the cache hierarchy. As a consequence of the resulting weakly-ordered memory consistency model, a fencing operation such as SFENCE should be used if multiple processors might use different memory types to read/write the memory location. See Section 2.4.3.2., “Caching of Temporal Vs. Non-Temporal Data” for further information about non-temporal stores.

MOVNTI—Move Integer Non-Temporal

Instruction	Description
MOVNI m32, r32	Move 32 bits from r32 to m32, minimizing pollution in the cache hierarchy.

Description

Moves a double word from the specified general-purpose register r32 to m32, minimizing pollution in the cache hierarchy. The linear address corresponds to the address of the least-significant byte of the referenced memory data. This store instruction minimizes cache pollution.

Operation

m32 = r32;

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.WNI(EDX bit 26) = 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
Interrupt 13	If any part of the operand lies outside the effective address space from 0 to 0FFFFH.
#NM	If TS bit in CR0 is set.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR (bit 9) = 0.
	If CPUID.WN(EDX bit 26) = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

Numeric Exceptions

None.

Comments

MOVNTI minimizes pollution in the cache hierarchy. As a consequence of the resulting weakly-ordered memory consistency model, a fencing operation such as SFENCE should be used if multiple processors may use different memory types to read/write the memory location. Note: Unlike store instructions supported in x87 numeric processor, MMX technology, single-precision, floating-point SIMD instructions in the Streaming SIMD Extension instructions and double-precision, floating-point SIMD instructions in the Streaming SIMD Extensions 2, this does not trigger a #NM (DNA exception). See Section 2.4.3.2., "Caching of Temporal Vs. Non-Temporal Data" for further information about non-temporal stores.

PAUSE—Pause For Preset Amount of Time

Instruction	Description
PAUSE	Delays execution of next instruction implementation-specific amount of time.

Description

Delays execution of the next instruction an implementation-specific amount of time. The delay is finite and can be zero for some processors. This instruction does not change architectural state (that is, it performs essentially a delaying no-op operation). This instruction is useful in situations where it is beneficial to moderate execution speed. For example, the PAUSE instruction can be used in the spin-wait loop of a routine implementing a lock. The PAUSE instruction can be inserted in the loop testing the status of the lock.

Operation

Execution of next instruction delayed implementation specfic amount of time.

Protected Mode Exceptions

None.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

None.

Numeric Exceptions

None.

3.5. MODIFIED INSTRUCTIONS

The Streaming SIMD Extensions 2 modify the behavior of the following IA-32 architecture instructions:

- CPUID instruction.
- RDPMC instruction.
- Branch instructions.

These modifications are described in the following sections.

3.5.1. CPUID Instruction

The CPUID instruction has been modified as functions:

- Operation of the CPUID instruction has been changed.
- New flags added to indicate the existence of Streaming SIMD Extensions 2.

The following pseudo code shows the operation of the CPUID instruction in Willamette family processor, depending on the value entered in the EAX register:

```
Switch(EAX)
case 0:
     EAX=X; # Highest value supported by the Willamette processor/*
Currently X=3 */
     EBX:EDX:ECX = "GenuineIntel"
     ;
case 1:
     EAX[3:0] = X /* Stepping ID = 0 */
     EAX[7:4] = X /* Model ID = 0 */
     EAX[11:8] = F /* Family = F */
     EAX[13:12] = 0 /* Processor Type = 0 */
     EAX[31:14] = reserved(0)
     EBX[7:0] = X /* Brand Index */
     EBX[15:8] = CLFLUSH chunk count
     EBX[23:16] = Reserved
     EBX[31:24] = X /*APIC ID assigned at hardware reset*/
     ECX[31:0] = Reserved
     EDX = Feature_flags (see Figure 7 below)
;
case 2:
     EAX[7:0] = Number_Param_Descrip_Blocks = 1
     EAX[31:8] = 0x665b50 /* cache and TLB parameters. */
     EBX, ECX = Reserved 0
     EDX = 0x007A7023 /* cache and TLB parameters. */
```

;

Table 3-2 shows the complete list of feature flags returned in the EDX register when the CPUID instruction is executed on a Willamette processor.

Table 3-2. Feature Flags Returned in EDX Register

Bit #	Mnemonic	Setting	Description
31-30	Reserved	0	Reserved.
29	Reserved	1	Reserved.
28	Reserved	X	Reserved.
27	SLFSNP	1	Self-Snoop. The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus
26	SSE-2	1	The processor supports the Streaming SIMD Extensions 2 to the IA-32 architecture.
25	SSE	1	The processor supports the Streaming SIMD Extensions to the IA-32 architecture.
24	FXSR	1	Indicates whether the processor supports the FXSAVE and FXRSTOR instructions for fast save and restore of the floating point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it uses the fast save/restore instructions
23	MMX	1	The processor supports the MMX technology extensions to the IA-32 architecture.
22	Reserved	1	Reserved.
21	DTES	1	Debug Trace Store and Event Montior support. The processor has the ability to write a history of the taken branch to and from addresses or precise execution statistics into a memory based buffer.
20	Reserved	0	Reserved.
19	CLFSH	1	CLFLUSH Instruction. CLFLUSH instruction is supported.
18	PN	X	Processor Number. The processor supports the 96-bit Processor Number feature, and the feature is enabled.
17	PSE	1	Indicates whether the processor supports 4-Mbyte pages that are capable of addressing physical memory beyond 4 Gbytes. This feature indicates that the upper four bits of the physical address of the 4-Mbyte page is encoded by bits 13-16 of the page directory entry.
16	PAT	1	Indicates whether the processor supports the Page Attribute Table. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory on a 4K granularity through a linear address.
15	CMOV	1	Conditional Move Instructions. The conditional move instruction CMOV is supported. Furthermore, if floating point is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported

Table 3-2. Feature Flags Returned in EDX Register

14	MCA	1	Machine Check Architecture. The Machine Check Architecture, which provides a compatible mechanism for error reporting in P6 and future processors, is supported. MCG_CAP itself contains feature bits describing how many banks of error reporting MSRs are supported.
13	PGE	1	PTE Global Bit. The global bit in Page Directory Entries (PDEs) and Page Table Entries (PTEs) is supported, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.
12	MTRR	1	Memory Type Range Registers. Memory Type Range Registers are supported. MTRRcap itself contains feature bits describing what memory types are supported, how many variable MTRRs are supported, and whether the fixed MTRRs are supported.
11	SEP	1	SYSENTER/SYSEXIT If SEP=1, then the CPU supports the SYSENTER and SYSEXIT instructions and associated MSRs.
10	Reserved	0	Reserved
9	APIC	1	APIC On-chip. The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated).
8	CX8	1	CMPXCHG8B Instruction. The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).
7	MCE	1	Machine Check Exception. Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature.
6	PAE	1	Physical Address Extension. Physical addresses >32 bits are supported: extended PTE formats, an extra level in the page translation tables is defined, 2Mbyte pages are supported instead of 4Mbyte pages if PAE=1. The actual number of address bits beyond 32 is not defined, and is implementation specific.
5	MSR	1	Model Specific Registers RDMSR and WRMSR instructions exist. Some of the actual registers are implementation dependent.
4	TSC	1	Time Stamp Counter. The RDTSC instruction is supported, including CR4.TSD for controlling privilege.
3	PSE	1	Page Size Extension. Large pages of size 4Mbyte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.
2	DE	1	Debugging Extensions. Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.

Table 3-2. Feature Flags Returned in EDX Register

1	VME	1	Virtual 8086 Mode Enhancements. Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.
0	FPU	1	Floating Point Unit On-Chip. The processor contains an FPU supporting the x87 FPU instruction set.

3.5.2. RDPMC Instruction

Willamette processors support both “fast” and “slow” reads of the performance counters via the RDPMC instruction. Bit 31 of ECX, if set, will cause the RDPMC instruction to perform a read of the low 32 bits of the performance counter that is addressed in ECX[31:0]. The 32-bit counter result will be returned in EAX and EDX will be set to zero. The 32-bit read will execute faster on the Willamette processor than the full 40-bit read. If Bit 31 of ECX is clear then RDPMC will execute a full 40-bit read of the performance counter addressed in ECX[31:0] with data returned in EDX:EAX.

3.5.3. Branch Hints

The Streaming SIMD Extensions 2 instructions have enhanced the behavior of branch instructions to allow the user to supply branch direction hints by means of instruction prefixes. Prefixes will be supported on branching instructions that provide the hint semantics listed in Table 3-3.

Table 3-3. Branch Hints

Hint	Description
HWNT	Hint weakly not taken
HST	Hint strongly taken

CHAPTER 4
CODE OPTIMIZATION

This chapter provides some guidelines for optimizing code for the Willamette family of processors. In its current form, guidelines are provided without much detail. Later versions of this chapter will flesh out some justification for these guidelines.

The intended audience for this chapter includes compiler writers, assembly language programmers, high-level language programmers, and operating system programmers.

4.1. CODE OPTIMIZATION GUIDELINES

This section describes guidelines for code optimizations that are can be used to maximize the performance of Willamette processors.

4.1.1. Improve Branch Predictability

Use the following guidelines to improve branch prediction.

- Avoid unnecessary branches
  - Arrange basic blocks that form the most likely sequence so that they are contiguous in memory.
  - Unroll loops to eliminate branches if the number of iterations is known.
- Avoid misprediction penalties
  - Arrange code and set branch conditions so that the target of a backwards conditional branch is on the most likely path.
  - Arrange code and set branch conditions so that the code following a forward conditional branch is on the most likely path.
  - For loops with a small number of iterations, exit the loop using a forward conditional branch and follow the loop back edge using an unconditional branch.
  - Convert indirect jumps which have a directional bias into a sequence of conditional branches or predictable indirect branches. Mispredicted indirect branches are likely to have the same cost as mispredicted conditional branches, so this optimization should not be applied to indirect branches with evenly-weighted targets.
  - Make use of branch hints.

- Avoid implementing a call by pushing the return address and jumping to the target. The hardware can pair up call and return instructions to enhance predictability.

4.1.2. Scheduling

Use the following scheduling guidelines to maximize the flow of instructions through the processor:

- Trace cache scheduling guidelines
  - When possible, space branches out by an average of 2 intervening uops.
  - Let **IBR** stand for the class of branches that includes CALL, RET, IRET, indirect JMP and far transfers.
  - Avoid scheduling a branch less than 5 uops after an **IBR.**
  - Avoid scheduling a CISC less than 2 uops after an **IBR** or CISC instruction.
  - Avoid scheduling a FXCH less than 2 uops after an **IBR**, a CISC instruction, or another FXCH.
- Calculate store addresses as early as possible to avoid having stores block loads.

4.1.3. Code Selection

Use the following coding guidelines improve processor performance:

- The template restrictions that the P6 family processors issue no longer apply with Willamette family processor; however, there is still an overhead for CISC instructions.
- Avoid more than one instruction prefix.
- The LEA instruction tends to be more efficient in terms of encoding and latency than an equivalent set of macro instructions, except if the target register or address is 16 bits.
- Avoid instructions that contain both an immediate and a displacement.
- Avoid changing the segment registers.
- Avoid sign-extended moves from memory or to 16-bit registers.
- Avoid referencing the high-byte general-purpose registers (such as AH) since referencing them generally requires a shift.
- If there is more than one PUSH or POP instruction, use loads or stores and adjust the stack pointer once.
- The latencies of some integer instructions, e.g. shifts and multiplies, changed with respect to P6 family processors, which has implications on whether to emit a series of adds in place of a shift, or a series of shifts and adds in place of a multiply. Using a series of add can be more advantageous because add instruction has one half clock latency.

- The FXCH instruction has 0 latency, but is subject to issue restrictions. FXCH should be used only where it is expressly needed by the algorithm.
- Take advantage of hardware register renaming: do not worry about false Write-after-Write and Write-after-Read dependencies, which are handled by the hardware.

4.1.4. Memory

Use the following guidelines improve processor performance on memory accesses:

- Avoid misaligned memory accesses.
- Eliminate redundant loads and stores.
- For integer stores, avoid scaling and using multiple registers or ESP in the addressing mode. This enables mapping to single-uop stores.
- Avoiding breaking a single load into smaller pieces, e.g. loading 16 bits with two 8-bit loads.
- Avoid writing an operand of one size and then reading a larger operand from the same location.
- If possible, let a store and a following load to the same address have two other intervening loads to a different address. This has implications for object-oriented code. Rather than pushing and popping in the reverse order, stores and loads should be made asymmetric.
- Block caches to optimize for locality. If there are two threads which are using the shared cache, block for half-sized caches.

4.1.5. General Principles for Code and Data Layout

Use the following general principles for regarding code and data layout maximize processor performance:

- Do not put code and data on the same 4K page.
- Put code with temporal and spatial locality in the same line and on the same page, move other code elsewhere, even to another page.
- Lay out probable sequences of basic blocks contiguously to optimize prefetch.
- Align branch targets on a 16B boundary, if possible. There may be some additional benefit to aligning branch targets to 64B boundaries.
- Align words, doublewords, quadwords, and double quadwords on 2-, 4-, 8-, and 16-byte boundaries, respectively. Consider alignment issues on the stack as well as for static data, whether by explicit manipulation of SP or of a pointer to stack data which must be aligned.

4.1.6. Make Use of Prefetching

Use the following guidelines for data prefetching to improve processor performance:

- Target time-consuming innermost loops, loads at the beginning of a critical dependence chain.
- Schedule the prefetch far enough ahead to cover the expected latency from main memory.
- Do not place prefetches too far ahead, otherwise pollution will result.
- Prefetch to the appropriate cache level. If fetching far ahead, prefetch to a higher level cache, making sure the capacity is there.
- Do not prefetch excessively. Prefetching uses up instruction, memory and bus bandwidth. Prefetched lines will occupy the load buffer resource, and instruction issue may be stalled because of a lack of this resource.
- Intersperse prefetch instructions with other computational instructions to increase potential ILP, spread out burden on the load buffer and to allow potential dirty writebacks to proceed concurrently with other instructions.
- Use strip mining and loop blocking to enhance locality, thus reducing the total number of prefetches required.
- Use non-temporal prefetch for data which can be replaced soon. This enhances the cache-ability of other data since the non-temporally-prefetched data is less likely to replace it.
- Do not prefetch code using the prefetch instructions.

4.1.7. New Instructions

Use the following instruction selection guidelines to maximize processor performance:

- Maximize the amount of SIMD-type parallelism exploited, especially for vectorizable loops.
- When possible, use the Streaming SIMD Extensions 2 floating-point instructions instead of corresponding x87 FPU instructions. The Streaming SIMD Extensions 2 floating-point instructions have a shorter latency.
- Use the PAUSE instruction to temporarily suspend an operating system thread during synchronization

4.1.8. Code Size

The following guidelines for minimizing code size can be used to improve performance on all IA-32 processors:

- Using ESP in a mod/rm addressing mode uses an extra byte. Avoid this if possible.
- Using EAX instead of other registers for loading immediates saves a byte.

- For code that will be executed out of the trace cache, the decoder costs for bad code density or prefixes are amortized. Be more concerned with trace cache line packing constraints.

4.2. NOTABLE DIFFERENCES BETWEEN THE WILLAMETTE AND P6 FAMILY PROCESSORS

The following sections highlight some notable differences between Willamette and the P6 family processors that affect code performance.

4.2.1. Code Selection

- The decoder template restrictions found in the P6 family processor no longer apply with the Willamette family processors.
- Integer shifts are more expensive in Willamette family processors, and integer multiplies are somewhat more expensive.
- In Willamette family processors, the architected registers are renamed as a single unit, i.e. AL and AH reference different parts of the same register, and are not independent.
- Register partial stalls are no longer an issue in Willamette family processors.

4.2.2. New Instructions

- Make use of single and double precision floating point SIMD.
- Use Streaming SIMD Extensions 2 floating-point scalar instructions instead of corresponding x87 FPU instructions.
- Use prefetching (see Section 4.1.6., “Make Use of Prefetching”).

APPENDIX A STREAMING SIMD EXTENSIONS 2 INSTRUCTION SUMMARY

This appendix provides an alphabetical list of the instructions added to the IA-32 architecture with the Streaming SIMD Extensions 2. Given for each instruction is the assembly language mnemonic and instruction format, a description of the operation performed, and the SIMD floating-point exceptions that can be generated. The list is divided into three tables:

- Packed and scalar double-precision floating-point instructions.
- SIMD integer instructions.
- Cacheability control and memory ordering instructions.

Table A-1. Packed and Scalar Double-precision Floating-point Instructions

Instruction	Description	#I	#D	#Z	#O	#U	#P
ADDPD xmm1, xmm2/m128	Add packed double-precision floating-point numbers from xmm2/mem to xmm2.	Y	Y		Y	Y	Y
ADDSD xmm1, xmm2/m64	Add the low double-precision floating-point number from xmm2/mem64 to xmm1.	Y	Y		Y	Y	Y
ANDNPD xmm1, xmm2/m128	Invert the 128 bits of xmm1, then bitwise AND the result with the 128 bits of xmm2/mem128.						
ANDPD xmm1, xmm2/m128	Bitwise logical AND of xmm2/mem128 and xmm1.						
CMPPD xmm1, xmm2/m128, imm8	Compare packed double-precision floating-point numbers from xmm2/mem128 with packed double-precision floating-point numbers in xmm1, using imm8 as predicate.	Y	Y				
CMPSD xmm1, xmm2/m64, imm8	Compare low double-precision floating-point number from xmm2/mem64 with low double-precision floating-point number in xmm1 register using imm8 as predicate	Y	Y				

Table A-1. Packed and Scalar Double-precision Floating-point Instructions

Instruction	Description	#I	#D	#Z	#O	#U	#P
COMISD xmm1, xmm2/m64	Compare low double-precision floating-point number in xmm1 register with low double-precision floating-point number in xmm2/mem64 and set the EFLAGS flags register accordingly.	Y	Y				
CVTDQ2PD xmm1, xmm2/m64	Convert two packed doubleword signed integers from xmm2/m128 to two packed double-precision floating-point numbers.						
CVTPD2DQ xmm1, xmm2/m128	Convert two packed double-precision floating-point numbers from xmm2/m128 to two packed doubleword signed integers in xmm1 using rounding specified by MXCSR.	Y					Y
CVTPD2PI mm, xmm/m128	Convert two packer double-precision floating-point numbers from xmm/m128 to two packed doubleword signed integers in mm using rounding specified by MXCSR.	Y					Y
CVTPD2PS xmm1, xmm2/m128	Convert two double-precision floating-point numbers to two single-precision floating-point numbers.	Y	Y		Y	Y	Y
CVTPI2PD xmm1, mm2/m64	Convert lowest two signed doubleword integers from mm/mem64 to two double-precision floating-point numbers.						
CVTPS2PD xmm1, xmm2/m64	Convert two single-precision floating-point numbers to two double-precision floating-point numbers.	Y	Y				
CVTSD2SI r32, xmm/m64	Convert one double-precision floating-point number from xmm/m64 to one doubleword signed integer using the rounding mode specified by MXCSR, and return the result to a general-purpose register.	Y					Y
CVTSD2SS xmm1, xmm2/m64	Convert double-precision floating-point number in xmm2/m64 to single-precision floating-point number in xmm1.	Y	Y		Y	Y	Y

Table A-1. Packed and Scalar Double-precision Floating-point Instructions

Instruction	Description	#I	#D	#Z	#O	#U	#P
CVTSI2SD xmm, r/m32	Convert signed doubleword integer from r/m32 to a double-precision floating-point number.						
CVTSS2SD xmm1, xmm2/m128	Convert single-precision floating-point number to double-precision floating-point number.	Y	Y				
CVTTPD2DQ xmm1, xmm2/m128	Convert two double-precision floating-point numbers from xmm2/m128 to two signed doubleword integers in XMM using truncate.	Y					Y
CVTTPD2PI mm1, xmm2/m128	Convert two double-precision floating-point numbers from xmm2/m128 to two signed doubleword integers in mm using truncate.	Y					Y
CVTTSD2SI r32, xmm/m64	Convert one double-precision floating-point number from xmm/m64 to one signed doubleword integer using truncate, and return the result to r32.	Y					Y
DIVPD xmm1, xmm2/m128	Divide packed double-precision floating-point numbers in xmm1 by xmm2/mem128	Y	Y	Y	Y	Y	Y
DIVSD xmm1, xmm2/m64	Divide low double-precision floating-point numbers in xmm1 by xmm2/mem6	Y	Y	Y	Y	Y	Y
MAXPD xmm1, xmm2/m128	Return the maximum double-precision floating-point numbers between xmm2/mem128 and xmm1.	Y	Y				
MAXSD xmm1, xmm2/m64	Return the maximum double-precision floating-point number between the low double-precision floating-point numbers from xmm2/mem64 and xmm1.	Y	Y				
MINPD xmm1, xmm2/m128	Return the minimum double-precision floating-point numbers between xmm2/mem128 and xmm1.	Y	Y				
MINSD xmm1, xmm2/m64	Return the minimum double-precision floating-point number between the low double-precision floating-point numbers from xmm2/mem64 and xmm1.	Y	Y				

Table A-1. Packed and Scalar Double-precision Floating-point Instructions

Instruction	Description	#I	#D	#Z	#O	#U	#P
MOVAPD xmm1, xmm2/m128	Move packed double-precision floating-point numbers from xmm2/mem128 to xmm1.						
MOVAPD xmm2/m128, xmm1	Move packed double-precision floating-point numbers from xmm1 to xmm2/mem128.						
MOVHPD xmm, m64	Move double-precision floating-point number from memory to high quadword of XMM register.						
MOVHPD m64, xmm	Move double-precision floating-point number from high quadword of XMM register to memory.						
MOVLPD xmm, m64	Move double-precision floating-point numbers from memory to low quadword of XMM register.						
MOVLPD m64, xmm	Move double-precision floating-point numbers from low quadword of XMM register to memory						
MOVMSKPD r32, xmm	Move 2-bit mask to r32.						
MOVSD xmm1, xmm2/m64	Move scalar double-precision floating-point numbers from xmm2/m64 to xmm1 register.						
MOVSD xmm2/m64, xmm1	Move scalar double-precision floating-point numbers from xmm1 register to xmm2/m64.						
MOVUPD xmm1, xmm2/m128	Move two packed double-precision floating-point numbers from xmm2/mem128 to xmm1.						
MOVUPD xmm2/m128, xmm1	Move two packed double-precision floating-point numbers from xmm1 to xmm2/mem128						
MULPD xmm1, xmm2/m128	Multiply packed double-precision floating-point numbers in xmm2/mem128 by xmm1.	Y	Y		Y	Y	Y
MULSD xmm1 xmm2/m64	Multiply the low double-precision floating-point number in xmm2/mem64 by low double-precision floating-point number in xmm1.	Y	Y		Y	Y	Y
ORPD xmm1, xmm2/m128	Bitwise OR of xmm2/mem128 and xmm1.						
SHUFPD xmm1, xmm2/m128,imm8	Shuffle packed double-precision floating-point numbers.						

Table A-1. Packed and Scalar Double-precision Floating-point Instructions

Instruction	Description	#I	#D	#Z	#O	#U	#P
SQRTPD xmm1, xmm2/m128	Computes square roots of the packed double-precision floating-point numbers in xmm2/mem128.	Y	Y				Y
SQRTSD xmm1, xmm2/m64	Computes square root of the low double-precision floating-point number in xmm2/mem64.	Y	Y				Y
SUBPD xmm1 xmm2/m128	Subtract packed double-precision floating-point numbers in xmm2/mem128 from xmm1.	Y	Y		Y	Y	Y
SUBSD xmm1, xmm2/m64	Subtracts the low double-precision floating-point numbers in xmm2/mem64 from xmm1.	Y	Y		Y	Y	Y
UCOMISD xmm1, xmm2/m64	Compares (unordered) the low double-precision floating-point number in xmm1 register with the low double-precision floating-point number in xmm2/mem64 and set the EFLAGS register accordingly.	Y	Y				
UNPCKHPD xmm1, xmm2/m128	Interleaves double-precision floating-point numbers from the high quadwords of xmm1 and xmm2/mem128.						
UNPCKLPD xmm1, xmm2/m128	Interleaves double-precision floating-point numbers from the low quadwords of xmm1 and xmm2/mem128.						
XORPD xmm1, xmm2/m128	Bitwise XOR of xmm2/mem128 and xmm1.						

Table A-2. SIMD Integer Instructions

Instruction	Description	#I	#D	#Z	#O	#U	#P
CVTDQ2PS xmm, xmm/m128	Convert four signed doubleword integers to four single-precision floating-point numbers.						Y
CVTPS2DQ xmm, xmm/m128	Convert four packed single-precision floating-point numbers from xmm2/m128 to four packed doubleword signed integers in xmm1 using rounding specified by MXCSR.	Y					Y
CVTTPS2DQ xmm, xmm/m128	Convert four single-precision floating-point numbers from xmm2/m128 to four doubleword signed integers in xmm1 using truncate.	Y					Y
MOVD xmm, r/m32	Move doubleword from general-purpose register or memory to XMM.						
MOVD r/m32, xmm	Move doubleword from XMM register to general-purpose register/memory.						
MOVDQA xmm1, xmm2/m128	Move aligned double quadword from xmm2/mem128 to xmm1.						
MOVDQA xmm2/m128, xmm1	Move aligned double quadword from xmm1 to xmm2/mem128.						
MOVDQU xmm1, xmm2/m128	Move unaligned double quadword from xmm2/mem128 to xmm1.						
MOVDQU xmm2/m128, xmm1	Move unaligned double quadword from xmm1 to xmm2/mem128.						
MOVDQ2Q mm1, xmm2	Move low quadword from XMM to MMX register.						
MOVQ2DQ xmm2, mm1	Move quadword from MMX register to low quadword of XMM.						
MOVQ xmm1, xmm2/m64	Move quadword from xmm2/mem64 to xmm1.						
MOVQ xmm2/m64, xmm1	Move quadword from xmm1 to xmm2/mem64.						
PACKSSWB xmm1, xmm2/m128	Pack signed words from xmm1 and xmm2/mem128 into signed bytes in xmm1, with signed saturation.						

Table A-2. SIMD Integer Instructions

Instruction	Description	#I	#D	#Z	#O	#U	#P
PACKSSDW xmm1, xmm2/m128	Pack signed doublewords from and xmm2/mem128 into signed words in xmm1, with signed saturation.						
PACKUSWB xmm1, xmm2/m128	Pack and saturate signed words from xmm1 and xmm2/mem128 into unsigned bytes in xmm1.						
PADDB xmm1,xmm2/m128	Add packed bytes from xmm2/mem128 to packed bytes in xmm1.						
PADDW xmm1, xmm2/m128	Add packed words from xmm2/mem128 to packed words in xmm1.						
PADDD xmm1, xmm2/m128	Add packed doublewords from xmm2/mem128 to packed doublewords in xmm1.						
PADDQ mm1,mm2/m64 PADDQ xmm1,xmm2/m128	Add quadword integers from mm2/Mem to mm1. Add packed quadword integers from XMM2 /Mem to packed quadword integers in xmm1 register						
PADDSB xmm1, xmm2/m128	Add packed signed byte integers from xmm2/mem128 to packed signed bytes in xmm1, with saturation.						
PADDSW xmm1, xmm2/m128	Add packed signed word integers from xmm2/mem128 to packed signed words in xmm1, with saturation.						
PADDUSB xmm1, xmm2/m128	Add packed unsigned byte integers from xmm2/mem128 to packed unsigned byte integers in xmm1, with saturation.						
PADDUSW xmm1, xmm2/m128	Add packed unsigned word integers from xmm2/mem128 to packed unsigned word integers in xmm1, with saturation.						
PAND xmm1, xmm2/m128	Bitwise AND of xmm2/mem128 and xmm1.						
PANDN xmm1, xmm2/m128	Bitwise AND NOT of xmm2/mem128 and xmm1.						
PAVGB xmm1,xmm2/m128	Average packed unsigned bytes from xmm2/mem128 and xmm1, with rounding.						

Table A-2. SIMD Integer Instructions

Instruction	Description	#I	#D	#Z	#O	#U	#P
PAVGW xmm1, xmm2/m128	Average packed unsigned words from xmm2/mem128 and xmm1, with rounding.						
PCMPEQB xmm1, xmm2/m128	Compare packed bytes in xmm2/mem128 to packed bytes in xmm1 for equality.						
PCMPEQW xmm1, xmm2/m128	Compare packed words in xmm2/mem128 to packed words in xmm1 for equality.						
PCMPEQD xmm1, xmm2/m128	Compare packed doublewords in xmm2/mem128 to packed doublewords in xmm1 for equality.						
PCMPGTB xmm1, xmm2/m128	Compare packed bytes in xmm1 with packed bytes in xmm2/mem128 for greater than.						
PCMPGTW xmm1, xmm2/m128	Compare packed words in xmm1 with packed words in xmm2/mem128 for greater than.						
PCMPGTD xmm1, xmm2/m128	Compare packed doublewords in xmm1 with packed doublewords in xmm2/mem128 for greater than.						
PEXTRW r32, xmm, imm8	Extract the word specified by imm8 from XMM and move it to a general-purpose register.						
PINSRW xmm, r32/m16, imm8	Move the low word of r32 or from m16 into xmm at the position specified by imm8.						
PMADDWD xmm1, xmm2/m128	Multiply the packed word integers in xmm1 by the packed word integers in xmm2/mem128, and add the adjacent doubleword results.						
PMAXSW xmm1, xmm2/m128	Return the maximum signed word integers between xmm2/mem128 and xmm1.						
PMAXUB xmm1, xmm2/m128	Return the maximum unsigned byte integers between xmm2/m128 and xmm1.						
PMINSW xmm1, xmm2/m128	Return the minimum signed word integers between xmm2/mem128 and xmm1.						
PMINUB xmm1, xmm2/m128	Return the minimum unsigned byte integers between xmm2/mem128 and xmm1.						

Table A-2. SIMD Integer Instructions

Instruction	Description	#I	#D	#Z	#O	#U	#P
PMOVMSKB r32, xmm	Move the byte mask of xmm to r32.						
PMULHW xmm1, xmm2/m128	Multiply the packed signed word integers in xmm1 by the packed signed word integers in xmm2/mem128, and store the high words of the results in xmm1.						
PMULHUW xmm1, xmm2/m128	Multiply the packed unsigned word integers in xmm1 by the packed unsigned word integers in xmm2/mem128, and store the high words of the results in xmm1.						
PMULLW xmm1, xmm2/m128	Multiply the packed signed word integers in xmm1 by the packed signed word integers in xmm2/mem128, and store the low words of the results in xmm1.						
PMULUDQ mm1, mm2/m64	Multiply unsigned doubleword integer in mm1 by unsigned doubleword integer in mm2/m64, and store the quadword result in mm1.						
PMULUDQ xmm1, xmm2/m128	Multiply packed unsigned doublewords in xmm1 by packed unsigned doublewords in xmm2/mem128, and then store the 64-bit results in xmm1.						
POR xmm1, xmm2/m128	Bitwise OR of xmm2/mem128 and xmm1.						
PSADBW xmm1,xmm2/m128	Absolute difference of packed unsigned byte integers from xmm2 /m128 and xmm1; these differences are then summed within separate high and low 64-bit sections to produce two word results.						
PSHUFD xmm1, xmm2/m128, imm8	Shuffle the doublewords in xmm2/mem128 based on the encoding in imm8 and store result in xmm1.						
PSHUFHW xmm1, xmm2/m64, imm8	Shuffle the high words in xmm2/mem128 based on the encoding in imm8 and store the result in xmm1.						

Table A-2. SIMD Integer Instructions

Instruction	Description	#I	#D	#Z	#O	#U	#P
PSHUFLW xmm1, xmm2/m64, imm8	Shuffle the low words in xmm2/mem128 based on the encoding in imm8 and store in xmm1.						
PSLLDQ xmm1, imm8	Shift left xmm1 by imm8 bytes, clearing low-order bits.						
PSLLW xmm1, xmm2/m128	Shift words in xmm1 register left by amount specified in xmm2/mem128, while shifting in zeros.						
PSLLW xmm1, imm8	Shift words in xmm1 left by imm8.						
PSLLD xmm1, xmm2/m128	Shift doublewords in xmm1 left by amount specified in xmm2/mem128, while shifting in zeros.						
PSLLD xmm1, imm8	Shift doublewords in xmm1 by imm8.						
PSLLQ xmm1, xmm2/m128	Shift quadwords in xmm1 left by amount specified in xmm2/mem128, while shifting in zeros.						
PSLLQ xmm1, imm8	Shift quadwords in xmm1 by imm8.						
PSRAW xmm1, xmm2/m128	Shift words in xmm1 right by amount specified in xmm2/mem128 while shifting in sign bits.						
PSRAW xmm1, imm8	Shift words in xmm1 right by imm8 while shifting in sign bits						
PSRAD xmm1, xmm2/m128	Shift doubleword in xmm1 right by amount specified in xmm2 /m128 while shifting in sign bits.						
PSRAD xmm1, imm8	Shift doublewords in xmm1 right by imm8 while shifting in sign bits.						
PSRLDQ xmm1, imm8	Shift right xmm1 by imm8, clearing high-order bits.						
PSRLW xmm1, xmm2/m128	Shift words in xmm1 right by amount specified in xmm2/mem128 while shifting in zeroes.						
PSRLW xmm1, imm8	Shift words in xmm1 right by imm8.						

Table A-2. SIMD Integer Instructions

Instruction	Description	#I	#D	#Z	#O	#U	#P
PSRLD xmm1, xmm2/m128	Shift doublewords in xmm1 right by amount specified in XMM2 /Mem while shifting in zeroes.						
PSRLD xmm1, imm8	Shift doublewords in xmm1 right by imm8.						
PSRLQ xmm1, xmm2/m128	Shift quadwords in xmm1 right by amount specified in xmm2/mem128 while shifting in zeroes.						
PSRLQ xmm1, imm8	Shift quadwords in xmm1 right by imm8.						
PSUBB xmm1, xmm2/m128	Subtract packed bytes in xmm2/mem from packed bytes in xmm2 register.						
PSUBW xmm1, xmm2/m128	Subtract packed words in xmm2/mem from packed words in xmm2 register.						
PSUBD xmm1, xmm2/m128	Subtract packed doublewords in xmm2 /mem from packed doublewords in xmm2 register.						
PSUBQ mm1,mm2/m64 PSUBQ xmm1,xmm2/m128	Subtract quadword from mm2 /m64 from quadword in mm1. Subtract quadwords from xmm2 /m128 from quadwords in xmm1.						
PSUBSB xmm1, xmm2/m128	Subtract packed signed byte integers in xmm2/mem128 from packed signed byte integers in xmm1 and saturate.						
PSUBSW xmm1, xmm2/m128	Subtract packed signed word integers in xmm2/mem128 from packed signed word integers in xmm1 and saturate.						
PSUBUSB xmm1, xmm2/m128	Subtract packed unsigned byte integers in xmm2/mem128 from packed unsigned byte integers in xmm1 and saturate.						
PSUBUSW xmm1, xmm2/m128	Subtract packed unsigned word integers in xmm2/mem128 from packed unsigned word integers in xmm1 and saturate.						
PUNPCKHBW xmm1, xmm2/m128	Interleave bytes from the high quadwords of xmm1 and xmm2/mem128 into xmm1.						

Table A-2. SIMD Integer Instructions

Instruction	Description	#I	#D	#Z	#O	#U	#P
PUNPCKHWD xmm1, xmm2/m128	Interleave words from the high quadwords of xmm1 and xmm2/mem128 into xmm1.						
PUNPCKHDQ xmm1, xmm2/m128	Interleave doublewords from the high quadwords of xmm1 and xmm2/mem128 into xmm1.						
PUNPCKHQDQ xmm1, xmm2/m128	Interleave high quadwords of xmm1 and xmm2/mem128 into xmm1						
PUNPCKLBW xmm1, xmm2/m128	Interleave bytes from the low quadwords of xmm1 and xmm2/mem128 into xmm1.						
PUNPCKLWD xmm2, xmm2/m128	Interleave words from the low quadwords of xmm1 and xmm2/mem128 into xmm1.						
PUNPCKLDQ xmm1, xmm2/m128	Interleave doublewords from the low quadwords of xmm1 and xmm2/mem128 into xmm1.						
PUNPCKLQDQ xmm1, xmm2/m128	Interleave the low quadwords of xmm1 and xmm2/mem128 into xmm1 register						
PXOR xmm1, xmm2/m128	Bitwise XOR of xmm2/mem128 and xmm1.						

Table A-3. Cacheability Control and memory Ordering Instructions

Instruction	Description
CLFLUSH r32/m8	Cache line containing r32/m8 is flushed and invalidated from all caches in the coherency domain.
LFENCE	Guarantees that every load instruction that precedes, in program order, the load fence instruction, is globally visible before any load instruction that follows the load fence, in program order, is globally visible.
MASKMOVDQU m128[edi], xmm1, mm2	Move 128 bits representing integer data from xmm1 to memory location specified by the EDI register, using the byte mask in xmm2.
MFENCE	Guarantees that every memory access that precedes, in program order, the memory fence instruction is globally visible before any memory instruction that follows the fence, in program order, is globally visible.
MOVNTPD m128, xmm	Move 128 bits representing packed double-precision floating-point data from xmm to m128, minimizing pollution in the cache hierarchy.
MOVNTDQ m128, xmm	Move double quadword representing integer operands (bytes, words, doublewords, quadwords) from xmm to m128, minimizing pollution within cache hierarchy.
MOVNTI m32, r32	Move 32 bits from r32 to m32, minimizing pollution in the cache hierarchy.
PAUSE	Execution of the next instruction is delayed an implementation specific amount of time.