

**Floating-Point Arithmetic with the Intel[®] Architecture
(IA) Floating-Point Unit (FPU), Streaming SIMD
Extensions (SSE) and Streaming SIMD Extensions 2
(SSE2)**

Version 2.0

7/00

Order Number: 248608-001

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Pentium® II processors, Pentium III processors, and Pentium 4 processors may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

†Third-party brands and names are the property of their respective owners.

Copyright © Intel Corporation 1999, 2000

Table of Contents

1.	Introduction.....	5
2.	Floating-Point Computations Using the IA FPU	6
2.1	The FPU Architecture.....	7
2.2	FPU Status Word and Control Word	8
2.3	Floating-Point Exceptions.....	10
2.4	Software Exception Handling	13
2.5	Operating on NaNs.....	14
2.6	FPU Instructions	15
2.7	Examples.....	19
3	Streaming SIMD Extensions.....	32
3.1	Streaming SIMD Extensions Architecture.....	32
3.2	SIMD Control and Status Register.....	33
3.3	Floating-Point Exceptions.....	34
3.4	Software Exception Handling	36
3.5	Operating on NaNs.....	36
3.6	Streaming SIMD Extensions.....	37
3.7	Writing Applications with SSE Instructions.....	39
3.8	Examples.....	39
4	Streaming SIMD Extensions 2 Instructions.....	43
4.1	Streaming SIMD Extensions 2 Architecture.....	43
4.2	SIMD Control and Status Register.....	43
4.3	Floating-Point Exceptions.....	44
4.4	Software Exception Handling	45
4.5	Operating on NaNs.....	45
4.6	SSE2 Instructions	45
4.7	Writing Applications with SSE2 Extensions	48
4.8	Example	48
5	Summary of Differences	50
6	Conclusion	58

Revision History

Revision	Revision History	Date
2.0	Updated for the Pentium® 4 Processor	7/00
1.0	Original publication of document	9/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

- [1] IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985
- [2] Intel Corporation, *Intel® Architecture Software Developer's Manual*, vol. 1, 2, 3, 1999
- [3] Microsoft Corporation, *Visual C++† 6.0 On-Line Manual*, 1999

Abstract

This document gives an overview of the Intel® Architecture Floating-Point Unit (FPU) and floating-point instructions (x87 instructions) and of the new floating-point additions to the Intel architecture processors: the Streaming SIMD Extensions (SSE) and the Streaming SIMD Extensions (SSE2). Specific features, advantages, and differences with respect to traditional x87 floating-point computations, and the level of support of the IEEE Standard for Binary Floating-Point Arithmetic are described.

1. Introduction

Floating-point computations on Intel architecture (IA) processors prior to the introduction of the Pentium® III processor could be performed only with scalar operations, using the IA floating-point stack model. New applications that use 3D graphics, video decoding/encoding, or speech recognition algorithms require increasingly higher performance, while allowing for the same floating-point operation to be performed on multiple data elements in parallel. This corresponds to the SIMD (single instruction, multiple data) computation model offered by the SSE and the SSE2 extensions.

The SSE extensions accelerate performance of 3D graphics applications over previous Intel architecture processors. The programming model is similar to that for the MMX™ technology instructions model, but the SSE single precision floating-point instructions operate on packed single precision floating-point numbers rather than packed integer numbers. The SSE extensions also include some extensions to the 64-bit SIMD integer instruction set, as well as a few single precision floating-point scalar instructions.

The SSE2 technology introduces new SIMD double precision floating-point instructions and new SIMD integer instructions into the IA-32 Intel architecture. In particular, the SIMD double precision floating-point instructions accelerate the performance of scientific applications that require higher precision than single precision. Furthermore, the 128-bit SIMD integer instructions in SSE2 enable new high performance applications requiring integers with higher dynamic ranges and can accelerate the performance of applications that were previously implemented using the 64-bit MMX technology. This application note focuses on several coding techniques that are useful when dealing with numerical precision issues associated with x87 floating-point computations, SSE single precision floating-point computations, and SSE2 double precision floating-point computations.

Section 2 of this application note covers the IA Floating-Point Unit (FPU): data types, architecture, control and status words, floating-point exceptions, software exception handling, and operating on NaNs. It also includes a brief description of the FPU instructions. Examples illustrate rounding errors and their effects (including double-rounding errors), tininess detection, raising unmasked faults or traps, and the differences between the IA-32 FPU computation model and the IEEE [1] model. This review of the FPU architecture and operation allows also for a comparison between floating-point computations performed using the IA FPU and computations using the new instructions that operate on packed single or double precision floating-point numbers. The reader who is familiar with the Intel Architecture Floating-Point Unit may skip this section. As SSE and SSE2 include only floating-point addition, subtraction, multiplication, division (including reciprocal approximation), square root (including reciprocal square root approximation), comparison, and conversions between various data formats, this section concentrates mostly on related FPU instructions.

Section 3 describes SSE extensions: data types, architecture, control and status registers, floating-point exceptions, software exception handling, operating on NaNs, and gives a brief description of the new instructions. Two examples illustrate computations using SSE.

Section 4 covers SSE2 extensions, and is organized similarly to Section 3.

Finally, Section 5 summarizes the parallels between computations with FPU, SSE, and SSE2, pointing out the differences that exist among them. An example of a simple computation performed in single precision with SSE instructions, double precision with SSE2 instructions, and in double-extended precision with FPU instructions allows a comparison of the accuracy that can be expected from each model.

2. Floating-Point Computations Using the IA FPU

Floating-point numbers are generally encoded as a concatenation of a sign bit, an exponent, and a significand. The normal floating-point numbers have a sign bit of 0 (positive) or 1 (negative), an exponent in the range $[E_{\min}, E_{\max}]$, and a significand in the range $[1, 2)$, with an implicit binary point past its most significant bit (which means its integer bit or J-bit is 1). In the actual encoding, the exponent is biased (by adding a positive bias to it) so that the encoded exponent is always positive, while still being within the given exponent range. The value of a normal floating-point number is:

$$f = (-1)^{\text{sign}} \cdot \text{significand} \cdot 2^{\text{exponent-bias}}$$

The value corresponding to the least significant bit position in a floating-point number is called unit-in-the-last-place, or *ulp*. For the floating-point number f above, the representation is:

$$1 \text{ ulp} = 0.0\dots01 \cdot 2^{\text{exponent-bias}} = 2^{\text{exponent} - \text{bias} - N + 1}$$

where N is the number of bits in the significand.

The IA FPU supports three floating-point formats that can be stored in memory: single precision, double precision, and double-extended precision (mandated or recommended by the IEEE Standard for Binary Floating-Point Arithmetic [1]). Their characteristics are given in Table 1. In the FPU, floating-point numbers always are represented using 80 bits, with the exponent range for single and double precision numbers extended to 15 bits. Their significand size remains unchanged, but is padded with 0's up to the length of 64 bits, corresponding to the IA-32 stack single and double precision formats. Double-extended precision numbers have the same representation in the FPU and in memory. Note that in memory format, the integer bit is not explicitly represented for single and double precision range floating-point numbers.

Table 1: Characteristics of the single, double, IA-32 FP register stack single, IA-32 FP register stack double, and double-extended precision floating-point formats

Floating-Point Format	Single Precision (Memory Format)	Double Precision (Memory Format)	IA-32 Stack Single Precision	IA-32 Stack Double Precision	Double-Extended Precision
Exponent Bits	8	11	15	15	15
Significand Bits	24	53	24	53	64
Format Size (bits)	32	64	80 (40 trailing 0's)	80 (11 trailing 0's)	80
E_{\min}	-126	-1022	-16382	-16382	-16382
E_{\max}	127	1023	16383	16383	16383
Exponent Bias	127	1023	16383	16383	16383
Format Range (Absolute Val.)	2^{-149} to $(2 \cdot 2^{-23}) \cdot 2^{127}$	2^{-1074} to $(2 \cdot 2^{-52}) \cdot 2^{1023}$	2^{-16405} to $(2 \cdot 2^{-23}) \cdot 2^{16383}$	2^{-16434} to $(2 \cdot 2^{-52}) \cdot 2^{16383}$	2^{-16445} to $(2 \cdot 2^{-63}) \cdot 2^{16383}$
Format Range (Approximate Absolute Val.)	$10^{-44.85}$ to $10^{38.53}$	$10^{-323.3}$ to $10^{308.2}$	10^{-4938} to 10^{4932}	10^{-4947} to 10^{4932}	10^{-4950} to 10^{4932}

Besides normal floating-point numbers, the following can be encoded (with the leading bit explicitly represented only in double-extended or floating-point register format):

- **denormal numbers:** exponent of 0, denoting actually the value of E_{\min} and a non-zero significand with a leading bit of 0
- **zero:** sign bit of 0 or 1, exponent of 0, and significand of 0
- **infinity:** sign bit of 0 or 1, exponent of 11...1, and significand of 10...0
- **NaNs (Not a Number):** sign bit ignored, exponent of 11...1, and non-zero significand different from 10...0 (which is reserved for infinity); NaNs have no floating-point numerical value associated with them; if the first fraction bit is 0, the NaN is a signaling NaN (SNaN, used to trigger invalid operation floating-point exceptions); if the first fraction bit is 1, the NaN is a quiet NaN (QNaN); a special instance of this type is the QNaN Real Indefinite value (sign=1, exponent=11...1, significand=110...0), that is returned as a result for certain invalid operations (such as $\infty - \infty$)

In double-extended or FPU stack format, there are certain floating-point numbers whose encoding is not supported by the Intel architecture:

- unnormal floating-point numbers, characterized by a non-zero exponent different from 11...1 and a significand with a leading bit of 0
- pseudo-infinities, with an exponent of 11...1 and a significand of zero
- pseudo-NaN, with an exponent of 11...1 and a non-zero significand with a leading bit of 0

The unsupported encodings are only possible for double-extended or FPU stack format numbers (since the J-bit equal to 1 is implicit in the single and double precision memory formats).

2.1 The FPU Architecture

The IA FPU is a coprocessor that operates in parallel with the processor's integer unit. The actual microarchitecture varies with the generation of processor. Up to two integer units and two floating-point units are present in a given 32-bit implementation of the Intel architecture to this date.

The FPU contains eight 80-bit data registers that can be loaded from memory, or can be the destination of a floating-point instruction result. If the format of a value loaded from memory is not double-extended precision (single precision floating-point, double precision floating-point, integer, or packed BCD integer), the value is automatically converted to double-extended precision.

The FPU data registers are organized as a circular stack. The register number of the current top-of-stack register (containing the latest data item placed on the stack) is stored in the TOP field of the FPU status word as shown in Figure 2. The register on top of the stack is always referred to as ST(0). The following registers are ST(1), ST(2), ... ST(7). A new value is pushed on the stack in ST(0), and the previous ST(0) becomes ST(1), ST(1) becomes ST(2), and so on. Stack overflow occurs if all the eight slots are full. When a value is popped off the stack from ST(0), the previous ST(1) becomes ST(0), ST(2) becomes ST(1), and so on. Stack underflow occurs if the stack is empty. The FXCH instruction allows swapping the top-of-stack register and any other register in the stack at very low cost, thus eliminating an inherent bottleneck associated with the stack model.

The FPU execution environment consists of the following registers, operation code, and pointers:

- eight FPU Data Registers
- a Control Register containing the Control Word

- a Status Register containing the Status Word
- a Tag Register indicating whether individual FPU registers contain valid data, zero, a special value, or are empty
- the floating-point Operation Code (11-bit opcode of the last non-control instruction executed by the FPU)
- the FPU Instruction Pointer (48-bit pointer to the last non-control instruction executed by the FPU)
- the FPU Operand (Data) Pointer (48-bit pointer to the data – if any – of the last non-control instruction executed by the FPU).

The processor saves the FPU Operation Code and Instruction and Operand (Data) in order to provide state information to exception handlers (software exception handlers have an operating system component, and a user component, invoked by the operating system).

Note that the MMX technology instructions use the MMX technology registers that are mapped to the FPU registers. Transitioning from MMX technology operations to FPU operations requires using the EMMS instruction, which sets the FPU tag word to empty (1's in all tag fields), marking the MMX technology registers as available. The opposite transition, from FPU to MMX technology operations does not require executing special instructions, as the execution of any MMX technology instruction sets the FPU tag word to full (0's in all tag fields), and the TOP to 0. These changes are architecturally visible if TOP was not zero, or if the tags were not all zero. In order to avoid unintended loss of information, it is recommended that before a transition, the FPU operations leave the FPU stack empty. Also, MMX technology code should not leave values in registers with the intention of using them later, if FPU operations are possible in between.

2.2 FPU Status Word and Control Word

The 16-bit FPU Control Word shown in Figure 1 controls masking and unmasking of floating-point exceptions, the precision mode, and the rounding mode. Bits 0 through 5 (IM, DM, ZM, OM, UM, PM) are the exception mask bits. If set, they mask (disable) the FPU exceptions (invalid operation, denormal, divide-by-zero, overflow, underflow, and inexact-result exceptions respectively). Bits 8 and 9 represent the precision control field (PC) that determines the precision of the floating-point calculations performed by the FPU. The significand of the result will have 24 bits if PC=00B, 53 bits if PC=10B, and 64 bits if PC=11B. The value PC=01B is reserved (in early IA processors, it was used to indicate the double-extended precision). The PC field affects only the floating-point add, subtract, multiply, divide, and square root operations. Bits 10 and 11 represent the rounding control field (RC) that determines which IEEE rounding method [1] is used for results that cannot be represented exactly in the destination format. Rounding is performed to nearest if RC=00B, down if RC=01B, up if RC=10B, and toward zero if RC=11B. Finally, bit 12 (X) is the Infinity Control Flag that has no current role and is only provided for backward compatibility reasons.

Rounding of a floating-point result is performed to the destination precision (the number of bits in the significand), and considering the bounded exponent range of the destination format. In some situations, it is useful to consider a hypothetical result obtained by rounding to the destination precision, but as though the exponent range were unbounded (as in [1], section 7.4).

The actual result provided by the hardware differs from the hypothetical result only if the latter is huge (its exponent is larger than E_{max}), or if it is tiny (the hypothetical result is non-zero and its exponent is smaller than E_{min}). If the hypothetical result is huge, the actual result will be (in absolute value) equal to ∞ or to the largest floating-point number that can be represented in the destination format (FPMAX =

$1.1\dots1 \cdot 2^{E_{\max}}$). If the hypothetical result is tiny, then the actual rounding has to be “undone” in order to avoid a double rounding error, and the result will be obtained through denormalization (shifting right the significand while inserting zeroes to the left, and incrementing the exponent until E_{\min} is reached), and then rounding to the destination precision. The result is equal in absolute value to the smallest normal number ($FPMIN = 1.0 \cdot 2^{E_{\min}}$), a denormal, or zero.

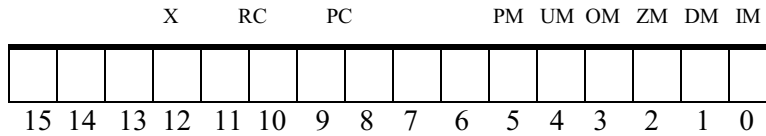


Figure 1: The FPU Control Word

The 16-bit FPU Status Word shown in Figure 2 indicates the current state of the FPU. Bits 0 through 5 (IE, DE, ZE, OE, UE, PE) are the exception flags for invalid operation, denormal, divide-by-zero, overflow, underflow, and inexact-result exceptions respectively. They indicate that one or more floating-point exceptions have been detected since they were last cleared. They are “sticky” bits, which means that once set, they will preserve their value until explicitly cleared. When set simultaneously with bit 0 (IE), bit 7 (SF) indicates a stack overflow (attempting to push a floating-point number on a full stack) or underflow (attempting to pop a floating-point number off an empty stack). When this occurs, bit 9 (C1) distinguishes between stack underflow ($C1 = 0$) and overflow ($C1 = 1$). Bit 7 (ES) is the exception summary bit that is set when any of the unmasked exception flags is set. Bits 8 through 10 and bit 14 (C0, C1, C2, C3) are the condition codes. They are used by certain floating-point comparison instructions to indicate their result (through C0, C2, and C3), and by certain arithmetic operations. For instance, when the PE flag is set indicating an inexact result, $C1 = 1$ indicates that the last rounding was upward. Other specialized roles of the condition codes are indicated in [2]. Bits 11,12, and 13 contain the TOP pointer to the top of the floating-point register stack. Bit 14 (B) indicates that the FPU is busy, and is maintained only for backward compatibility.

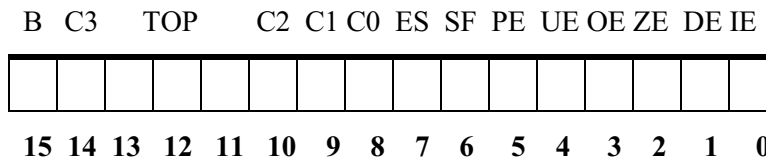


Figure 2: The FPU Status Word

Example 1: Tininess detection

The following computation illustrates tininess detection. It assumes that arithmetic operations can be performed in single precision in one step, as specified in [1]. Consider the multiplication of two single precision floating-point numbers a and b. The significands have 24 bits and are represented in binary, the exponents are represented in decimal, and $E_{\min} = -126$ is the smallest exponent for a normalized number:

$$a = 1.1\dots10 \cdot 2^{-126} \approx 2.35 \cdot 10^{-38}$$

$$b = 1.0\dots01 \cdot 2^{-1} \approx 0.5$$

Represented in decimal, the infinitely precise value of the product is:

$$a \cdot b = (2 - 2^{-22}) \cdot (1 + 2^{-23}) \cdot 2^{-127} = (2 - 2^{-45}) \cdot 2^{-127}$$

With an unbounded significand, the infinitely precise value of the product is:

$$a \cdot b = 1.11\dots1 \cdot 2^{-127}$$

where the significand is extended to 46 bits of 1. The hypothetical result obtained by rounding to 24 bits in the significand and using an unbounded exponent can be used to evaluate tininess. The hypothetical result depends on the rounding mode (the significands below have 24 bits):

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ for rounding to nearest (not tiny)}$$

$$a \cdot b = 1.11\dots1 \cdot 2^{-127} \text{ for rounding down (tiny)}$$

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ for rounding up (not tiny)}$$

$$a \cdot b = 1.11\dots1 \cdot 2^{-127} \text{ for rounding to zero (tiny)}$$

The actual rounding (to 24 bits in the significand and bounded exponent) depends also on the rounding mode (the significands below have 24 bits), yielding the rounded result below (denormal for rounding down and to zero; smallest normal for rounding to nearest and up):

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ for rounding to nearest} \quad (\text{set P flag})$$

$$a \cdot b = 0.11\dots1 \cdot 2^{-126} \text{ for rounding down} \quad (\text{set P,U flags})$$

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ for rounding up} \quad (\text{set P flag})$$

$$a \cdot b = 0.11\dots1 \cdot 2^{-126} \text{ for rounding to zero} \quad (\text{set P,U flags})$$

Example 4 in the Examples section (Section 2.7) includes sample code for this calculation.

2.3 Floating-Point Exceptions

The FPU detects six types of floating-point exceptions: invalid (#I) (subdivided into #IS - stack overflow or underflow, and #IA - invalid arithmetic operation), denormal operand (#D), divide-by-zero (#Z), overflow (#O), underflow (#U), and inexact-result, or precision (#P). Each exception has associated with it a mask/unmask (disable/enable) bit in the FPU control word, and a “sticky” exception status flag bit in the FPU status word.

Invalid operation, denormal operand, and divide-by-zero are pre-computation exceptions (floating-point faults), while overflow, underflow, and inexact-result are post-computation exceptions (floating-point traps).

If an exception is detected that is masked (mask bit set), the FPU handles the exception automatically, producing a predefined result and allowing program execution to continue.

If an exception is detected that is unmasked (mask bit clear), the FPU invokes a software exception handler to process the exception. For floating-point faults (#I, #D, #Z), the input operands of the excepting instruction are left unchanged, allowing the software handler to generate a result based on the input values. For floating-point traps, the input operands of the excepting instruction might have been overwritten, but a result is provided to the software handler that may further be used to generate a result.

In either case (masked or unmasked exception), the corresponding status flag in the FPU status register is set.

Invalid operation exceptions are caused by SNaN operands (or even QNaN for some floating-point compare instructions), by floating-point numbers in unsupported format (for example pseudo-infinities) or outside the admissible range (e.g. finite numbers that are too large to be converted to integer by the floating-point to integer conversion instruction). They are also caused by calculations such as $\infty - \infty$, $0 \cdot \infty$, $0 / 0$, ∞ / ∞ , or square root of a non-zero negative number (the square root of -0.0 is -0.0).

Denormal operand exceptions (the only category not defined by the IEEE Standard [1]) are caused by denormal operands.

Divide-by-zero exceptions occur when dividing a non-zero normal or denormal number by positive or negative zero.

Overflow exceptions occur when the result is huge. For FPU instructions whose destination is on the FPU stack, this is detected considering the precision (number of bits in the significand) indicated by the PC field from the FPU control word, and the 15-bit exponent range.

If the overflow exceptions are masked, the result (in absolute value) is ∞ or MAXFP, and the overflow and inexact-result status flags are set in the FPU status word.

If the overflow exceptions are unmasked, the result delivered to the software trap handler depends on the destination of the floating-point instruction that caused the overflow. If the destination is a memory location (which can happen only for a floating-point store), the source and the destination operands are left unchanged, the overflow flag is set in the FPU status register, and the software exception handler is invoked (see Example 8 in the following Examples section). If the destination is a floating-point register on the FPU stack, then the infinitely precise result scaled down (divided) by 2^{24576} is rounded to the destination precision, and the overflow status flag is set. The inexact-result status flag is set in this case only if the result was inexact. Condition code C1 in the FPU status register is set if the significand after rounding is larger than the infinitely precise significand, and it is cleared otherwise (C0 through C3 are not sticky bits). The software exception handler is invoked (see Example 9 in the Examples section).

Underflow exceptions follow different rules, depending on whether they are masked or not. But similar to overflow detection for the FPU instructions whose destination is on the FPU stack, underflow is detected considering the precision (number of bits in the significand) indicated by the PC field from the FPU control word, and the 15-bit exponent range.

If the underflow exceptions are masked, the underflow status flag in the FPU status word is set only if the result is tiny and inexact; the inexact-result status flag will also be set.

If the underflow exceptions are unmasked, the software handler is invoked if the result is merely tiny. The result delivered to the software trap handler depends on the destination of the floating-point instruction that caused the underflow. If the destination is a memory location (which can happen only for a floating-point store), the source and the destination operands are left unchanged, the underflow flag is set in the FPU status register, and the software exception handler is invoked. If the destination is a floating-point register on the FPU stack, then the infinitely precise result scaled up (multiplied) by 2^{24576} is rounded to the destination precision, and the underflow status flag is set. The inexact-result status flag is set in this case only if the result was inexact. Condition code C1 in the FPU status register is set if the significand of the delivered result is larger than the infinitely precise significand, and it is cleared otherwise. The software exception handler is invoked.

The precedence of the FPU floating-point exceptions is:

- invalid operation exception due to: floating-point stack underflow/overflow, operand of unsupported format or outside the admissible range, or SNaN operand (NaN for certain floating-point compare instructions)
- QNaN operand (not an exception, but handling QNaN operands has priority over lower priority exceptions)
- invalid operation exceptions not mentioned above, or divide-by-zero exception
- denormal operand exception (if masked, execution continues and a lower priority exception can occur as well)

- numeric overflow or underflow exceptions, possibly in conjunction with the inexact-result exception
- inexact-result exception

It should be noted that floating-point instructions can be executed in parallel with integer or system instructions. Concurrent execution can cause problems for floating-point exception handlers, due to the way floating-point exceptions are reported. When an unmasked floating-point exception occurs, the FPU stops further execution of the floating-point instruction, but only invokes the software exception handler on the next occurrence of a “waiting” floating-point or WAIT/FWAIT instruction (a “waiting” floating-point instruction is one that triggers a pending unmasked floating-point exception raised by a previous floating-point instruction, as opposed to the a “non-waiting” floating-point instruction, that does not do so). To prevent changes to the source and/or destination operands of a floating-point instruction that has caused an unmasked exception between the time it was detected and the time the software handler is invoked, an exception synchronizing instruction (any “waiting” floating-point instruction or WAIT/FWAIT instruction) must follow every floating-point instruction that might signal an unmasked exception. This has to happen at least for instructions that have memory operands - if the operands are on the floating-point stack, they cannot be changed by non-floating-point instructions.

Example 2: Unmasked underflow exceptions

The following computation illustrates raising an unmasked underflow exception. Consider again the multiplication of the two single precision floating-point numbers a and b from Example 1:

$$a = 1.1...10 \cdot 2^{-126} \approx 10^{-54.28}$$

$$b = 1.0...01 \cdot 2^{-1} \approx 0.5$$

Assume the underflow exceptions are unmasked, the inexact-result exceptions are masked, the precision control is set to single precision, and the destination of the multiplication is a 32-bit memory location (this operation can be implemented with two instructions, FMUL and FST; see also Example 5 below):

The result of a hypothetical rounding to 24 bits in the significand and unbounded exponent depends on the rounding mode, and is tiny only for rounding down and rounding to zero (as seen in Example 1). The actual result computed by FMUL (as an IA-32 stack single precision format number) is placed on the register stack, and is equal to that of the hypothetical rounding. An underflow exception is not raised because the 15-bit exponent range of the IA-32 register stack single precision format is not exceeded:

$$a \cdot b = 1.0...0 \cdot 2^{-126} \text{ for rounding to nearest}$$

$$a \cdot b = 1.11...1 \cdot 2^{-127} \text{ for rounding down}$$

$$a \cdot b = 1.0...0 \cdot 2^{-126} \text{ for rounding up}$$

$$a \cdot b = 1.11...1 \cdot 2^{-127} \text{ for rounding to zero}$$

The FST instruction converts the result to single precision and stores it to memory (32-bit single precision format).

If rounding down or to zero, FST detects the tiny input value, and raises an underflow exception. The P and U status flags are set, and the software exception handler is invoked. The source and the destination operands are left unchanged.

If rounding to nearest or up, the P status flag is set, and the result of $1.0 \cdot 2^{-126}$ is stored at the destination address. Note that in these two cases (rounding to nearest or up) if the inexact-result exceptions are unmasked, an inexact exception is raised (as explained below).

Inexact-result (precision) exceptions occur when the result of a floating-point operation represented in the destination format is different from the infinitely precise result, calculated with both the significand and the exponent range unbounded.

If the inexact-result exceptions are masked and an underflow or overflow exception does not occur, the inexact-result status flag in the FPU status register is set and the rounded result is stored in the destination operand.

If the inexact-result exceptions are unmasked and an underflow or overflow exception does not occur, the inexact-result status flag in the FPU status register is set, the rounded result is stored in the destination operand (just as above), and a software exception handler is invoked.

If an inexact-result exception occurs in conjunction with masked numeric overflow or underflow, the overflow/underflow and inexact-result status flags are set and the result is stored as described for the overflow/underflow exceptions. If the inexact-result exception is unmasked, the FPU also invokes the software exception handler.

If an inexact-result exception occurs in conjunction with unmasked numeric overflow or underflow and the destination operand is a floating-point register on the FPU stack, the overflow/underflow and inexact-result status flags are set, the result is stored as described for the unmasked overflow/underflow exceptions, and the FPU invokes the software exception handler.

If an inexact-result exception (unmasked or not) occurs in conjunction with an unmasked numeric overflow or underflow exception, and the destination operand is a memory location (which can happen only for a floating-point store), the inexact-result condition is ignored.

2.4 Software Exception Handling

Two different modes of operation for invoking a software floating-point exception handler are available: internal (or native) mode, and external, asynchronous pin oriented mode (also known as MS-DOS[†] compatibility mode). The CR0.NE bit controls which mechanism is active.

The internal mode is the recommended one, and is selected by setting the NE flag in control register CR0 (CR0.NE=1). If an unmasked floating-point exception occurs, the software handler is invoked through software interrupt vector 16 (#MF) immediately before executing the next “waiting” floating-point instruction (non-control FPU instruction or WAIT instruction), or before the next MMX technology instruction. In this mode, floating-point exception delivery is synchronous, but is deferred until the next “waiting” floating-point instruction or the next MMX technology instruction.

The external, asynchronous pin oriented mode (MS-DOS compatibility mode) is selected by clearing the NE flag in control register CR0 (CR0.NE=0). It is provided for compatibility with older floating-point coprocessor oriented systems. Because it involves an interaction of CPU pins and external hardware, it is platform specific. Its intended usage model is as follows. Unmasked floating-point exceptions cause the FERR# pin of the processor to be asserted (this is done at all times, irrespective of whether CR0.NE=1). The FERR# pin may be asserted immediately upon execution of the instruction that caused the unmasked floating-point exception, or may be delayed until the next “waiting” floating-point instruction or the next MMX technology instruction. This is processor dependent - for example on the i486 processor it is asserted immediately in nearly all cases, and portable software should not make any assumptions regarding this. In response to the FERR# pin being asserted, external hardware may generate an external interrupt request, or to assert the IGNNE# pin. Exactly how this is done is platform specific. If an unmasked floating-point exception is pending, upon execution of the next “waiting” floating-point instruction or the next MMX technology instruction, the processor will freeze until either an external hardware interrupt arrives, or until the IGNNE# pin is asserted. An external hardware

interrupt may be generated by an external programmable interrupt controller – PIC and may use the INTR# pin (with interrupt vector number provided by the external hardware), or the #NMI pin (with interrupt vector 2). Thus, in external mode the hardware interrupt associated with an unmasked floating-point exception is asynchronous: it may arrive at any time after the instruction that created the exception, before the next “waiting” floating-point instruction or the next MMX technology instruction. Although for particular CPU versions and instructions it may be possible to know precisely when the interrupt arrives, portable software should not rely on this knowledge and should be capable of handling the interrupt associated with the exception in a totally asynchronous manner. It is certain though that execution of a “waiting” floating-point instruction or MMX technology instruction will “drain” any pending floating-point exceptions, and so guarantees that subsequent code that does not execute floating-point instructions need not worry about asynchronous floating-point exceptions.

Typical operations performed by the software handler are to examine the stored FPU state information, to correct the condition that caused the exception, to clear the exception flags in the status word, and to return to the interrupted program in order to resume normal execution. Resuming execution is controlled by the operating system that will restore also the modified FPU state. Clearing the FPU status word flags is sometimes necessary (depending on the application), because unmasking a floating-point exception for which the status flag is set will cause an invocation of the software handler as explained above.

2.5 Operating on NaNs

As stated, QNaNs (quiet NaNs) do not cause floating-point exceptions for most floating-point instructions (exceptions are some of the floating-point compare instructions). Yet, handling QNaNs takes precedence over some floating-point exceptions, as seen above. For example, QNaN/0.0 is QNaN, and will not cause a divide-by-zero exception. Table 2 lists the rules for generating QNaN results when FPU instructions encounter QNaN or SNaN operands, or when a masked invalid operation exception occurs. The least frequent situation will probably be that when both operands of an FPU instruction are SNaN. At least one of them has to be on the FPU stack, and the only way to get an SNaN there is by using FRSTOR, which loads the FPU state (including the eight floating-point registers) from memory. SNaNs can be created in memory, but instructions other than FRSTOR would convert them to QNaN before moving them to the FPU stack, or generating a result that depends on them, on the FPU stack.

Note that NaNs do not have a numerical value associated with them, and no NaN is equal to anything else. Yet, one can talk about their sign, exponent, and significand, just in order to be able to classify NaNs by the contents of these bit fields. Also note that the behavior shown in Table 2, consisting of returning the NaN with the sign bit of 0 when the two source operands are NaNs of ‘opposite signs’, only applies to IA processors beginning with the Pentium Pro processor (early processors were not consistent in this regard).

Table 2. Rules for Generating QNaN Results for FPU instructions

Source Operands	QNaN Result
SNaN and QNaN	The QNaN source operand
Two SNaNs	The SNaN with the larger significand, converted to a QNaN ('quieted'), or the 'positive' SNaN converted to a QNaN if the significands are the same
Two QNaNs	The QNaN with the larger significand, or the 'positive' QNaN if the significands are the same
SNaN and a real value	The SNaN, converted to a QNaN (quieted)
QNaN and a real value	The QNaN source operand
No NaN operand, but an invalid operation exception is signaled	QNaN Real Indefinite

2.6 FPU Instructions

The Intel architecture FPU instructions and the floating-point exceptions they might raise are enumerated and described briefly in this subsection. Some instructions allow versions that pop values off the floating-point register stack, or reverse the order of the operands. The subsection dedicated to examples that follows will pay more attention to FPU instructions that are similar in role to the SSE and the SSE2 instructions. The FPU instructions that have no operands or do not change substantially their operands are referred to as non-arithmetic instructions, and can cause only invalid exceptions due to a stack overflow or underflow. The other FPU instructions can raise any of the six numeric exceptions (in addition to stack overflow or underflow), and are referred to as arithmetic instructions. Complete descriptions of all the FPU instructions are given in [2].

1. Data transfer instructions:

- **FLD**: floating-point load - push 32-bit, 64-bit, or 80-bit floating-point value from memory or 80-bit floating-point value from FPU stack register onto the FPU register stack, in ST(0); floating-point exceptions: stack overflow, I, D
- **FST/FSTP**: floating-point store - store the value in the register stack top ST(0) to a 32-bit or 64-bit memory location, or to an 80-bit floating-point stack register; FSTP also allows storing the register stack top to 80-bit memory location and pops the register stack top in all cases; floating-point exceptions: stack underflow, I, O, U, P
- **FXCH**: exchange contents of stack register ST(i) and stack register top ST(0); floating-point exceptions: stack underflow
- **FCMOVcc**: floating-point conditional move from stack register ST(i) to stack register top ST(0), depending on the CF, ZF, PF bits in the EFLAGS register; floating-point exceptions: stack underflow
- **FILD**: load 16-bit, 32-bit, or 64-bit integer from memory into FPU stack register ST(0); floating-point exceptions: stack overflow

- FIST/FISTP: store the value in the register stack top ST(0) as integer in 16-bit, or 32-bit memory location; FISTP also allows storing the register stack top to a 64-bit memory location and pops the register stack top in all cases; floating-point exceptions: stack underflow, I, P
 - FBLD: convert 80-bit BCD value from memory to double-extended real format, and push onto FPU stack; floating-point exceptions: stack overflow
 - FBSTP: store contents of FPU stack register top ST(0) to 80-bit memory location, in BCD format, and pop ST(0); floating-point exceptions: stack underflow, I, P
2. Load constant instructions:
- FLDZ, FLD1, FLDPI, FLDL2T, FLDL2E, FLDLG2, FLDLN2: load respectively the values of +0.0, +1.0, π , $\log_2 10$, $\log_2 e$, $\log_{10} 2$, $\log_e 2$ onto the FPU stack in ST(0); floating-point exceptions: stack overflow
3. Basic arithmetic instructions:
- FADD/FADDP: floating-point add – add contents of stack register top ST(0) and 32-bit or 64-bit memory real, and push result on register stack; add contents of two stack registers (at least one has to be the top register), and replace either with the result; FADDP operates only on stack registers, and pops the register stack top; floating-point exceptions: stack underflow, I, D, O, U, P
 - FIADD: add 16-bit or 32-bit integer from memory (converted to double-extended format) to floating-point value on top of the FPU register stack ST(0); floating-point exceptions: I, D, O, U, P
 - FSUB/FSUBP/FSUBR/FSUBRP: floating-point subtract – FSUB/FSUBP are similar to FADD/FADDP (the stack register top ST(0) contains the first operand when a memory operand is involved); FSUBR/FSUBRP perform a floating-point reverse subtract, similar to FSUB/FSUBP, but with operands in reverse order; floating-point exceptions: stack underflow, I, D, O, U, P
 - FISUB/FISUBR: subtract integer (converted to double-extended format) from floating-point – similar to FIADD (the stack register top ST(0) contains the first operand and the destination); FISUBR is similar to FISUB, but with operands in reverse order; floating-point exceptions: I, D, O, U, P
 - FMUL/FMULP: floating-point multiply – similar to FADD/FADDP; floating-point exceptions: stack underflow, I, D, O, U, P
 - FIMUL: multiply floating-point and integer (converted to double-extended format) - similar to FIADD; floating-point exceptions: I, D, O, U, P
 - FDIV/FDIVP/FDIVR/FDIVRP: floating-point divide – FDIV/FDIVP are similar to FADD/FADDP (the stack register top ST(0) contains the dividend when a memory operand is involved); FDIVR/FDIVRP perform a floating-point reverse divide, similar to FDIV/FDIVP, but with operands in reverse order; floating-point exceptions: stack underflow, I, D, Z, O, U, P
 - FIDIV/FIDIVR: divide floating-point to integer (converted to double-extended format) – similar to FIADD (the stack register top ST(0) contains the dividend and the destination); FIDIVR is similar to FIDIV, but with operands in reverse order; floating-point exceptions: I, D, Z, O, U, P

- FSQRT: floating-point square root of the value on top of the register stack; floating-point exceptions: stack underflow, I, D, P
- FRNDINT: round to integer the floating-point value on top of the register stack, using the rounding mode from the FPU Control Word; floating-point exceptions: stack underflow, I, D, O, U, P
- FABS: absolute value of the floating-point number on top of the register stack; floating-point exceptions: stack underflow
- FCHS: change the sign of ST(0); floating-point exceptions: stack underflow
- FPREM: partial remainder – replace ST(0) with the remainder obtained from dividing ST(0) by ST(1) (the divide operation uses rounding to zero); floating-point exceptions: stack underflow, I, D, U
- FPREM1: IEEE partial remainder – replace ST(0) with the IEEE remainder [2] obtained from dividing ST(0) by ST(1) (the divide operation uses rounding to nearest); floating-point exceptions: stack underflow, I, D, U
- FXTRACT: separate the floating-point value in ST(0) into exponent and significand, store the exponent in ST(0), and push the significand onto the register stack (with original sign and exponent 0x3fff); floating-point exceptions: stack underflow, stack overflow, I, D, Z

4. Comparison and classification instructions:

- FCOM/FCOMP/FCOMPP: compare real - compare the value on top of the FPU register stack with a 32-bit real from memory, a 64-bit real from memory, or with another value on the FPU stack; FCOMP also pops the register stack top ST(0); FCOMPP allows comparing only the top two values on the FPU stack, and pops them off the stack when done; the result is stored in bits C3, C2, and C0 of the FPU Status Word; this is an “ordered” compare, which means that QNaN operands will cause invalid operation exceptions; floating-point exceptions: stack underflow, I, D
- FUCOM/FUCOMP/FUCOMPP: unordered compare real – similar to FCOM/FCOMP/FCOMPP, but do not allow comparing with memory operands, and QNaN operands do not cause invalid operation exceptions; floating-point exceptions: stack underflow, I, D
- FICOM/FICOMP: compare a 16-bit or 32-bit integer in memory with the floating-point value on top of the FPU stack; FICOMP also pops the register stack top ST(0); the result is stored in bits C3, C2, and C0 of the FPU Status Word; QNaN operands do not cause invalid operation exceptions; floating-point exceptions: stack underflow, I, D
- FCOMI/FCOMIP: compare the value on top of the FPU register stack with another value on the FPU stack and set EFLAGS; FCOMIP also pops the register stack top ST(0); QNaN operands cause invalid operation exceptions; floating-point exceptions: stack underflow, I
- FUCOMI/FUCOMIP: similar to FCOMI/FCOMIP, but QNaN operands do not cause invalid operation exceptions; floating-point exceptions: stack underflow, I
- FTST: compare contents of register stack top ST(0) and 0.0; the result is stored in bits C3, C2, and C0 of the FPU Status Word; floating-point exceptions: stack underflow, I, D

- FXAM: classify the contents of register stack top ST(0) as unsupported, NaN, zero, denormal, normal, infinity, or empty; the result is stored in bits C3, C2, and C0 of the FPU Status Word; floating-point exceptions: none

5. Trigonometric instructions:

- FSIN: replace ST(0) with its sine; floating-point exceptions: stack underflow, I, D, U, P
- FCOS: replace ST(0) with its cosine; floating-point exceptions: stack underflow, I, D, P (U cannot be raised)
- FSINCOS: replace ST(0) with its sine and push the cosine onto the FPU stack; floating-point exceptions: stack underflow, I, D, U, P
- FPTAN: tangent - replace ST(0) by $\tan(\text{ST}(0))$ and push 1.0 onto the FPU stack (for arguments less than 2^{63} in absolute value): floating-point exceptions: stack underflow, I, D, U, P
- FPATAN: arctangent - replace ST(1) by $\arctan(\text{ST}(1)/\text{ST}(0))$ and pop the register stack ST(0); floating-point exceptions: stack underflow, I, D, U, P

A value of PI with 66 bits in the significand (stored internally) is used for argument reduction when needed.

6. Logarithmic, exponential and scale instructions:

- FYL2X: replace ST(1) by $\text{ST}(1) * \log_2 \text{ST}(0)$ and pop ST(0); floating-point exceptions: stack underflow, I, D, Z, O, U, P
- FYL2XP1: replace ST(1) by $\text{ST}(1) * \log_2 (\text{ST}(0) + 1.0)$ and pop ST(0); floating-point exceptions: stack underflow, I, D, O, U, P
- F2XM1: replace ST(0) by $2^{\text{ST}(0)} - 1$; floating-point exceptions: stack underflow, I, D, U, P
- FSCALE: scale ST(0) by ST(1); floating-point exceptions: stack underflow, I, D, O, U, P

7. FPU control instructions (no floating-point exceptions raised unless noted otherwise):

- FINIT/FNINIT: initialize FPU to round to nearest, 64-bit precision and exceptions masked, after (for FINIT) or without (for FNINIT) checking for pending unmasked floating-point exceptions
- FLDCW: load FPU control word from 2-byte memory location; this might unmask an exception pending in the FPU status word, that will be generated upon the execution of the next “waiting” FPU instruction
- FSTCW/FNSTCW: store FPU control word into 2-byte memory location after (for FSTCW) or without (for FNSTCW) checking for pending unmasked floating-point exceptions
- FSTSW/FNSTSW: store FPU status word into 2-byte memory location or into AX register after (for FSTSW) or without (for FNSTSW) checking for pending unmasked floating-point exceptions
- FCLEX/FNCLEX: clear floating-point exception flags after (for FCLEX) or without (for FNCLEX) checking for pending unmasked floating-point exceptions
- FLDENV: load FPU environment from 14-byte or 28-byte memory area (depending on the processor operating mode); if a status flag set to 1 is loaded in the FPU status word for an

exception that is unmasked, the exception will be generated upon execution of the next “waiting” floating-point instruction

- FSTENV/FNSTENV: store FPU environment into 14-byte or 28-byte memory area (depending on the processor operating mode) after (for FSTENV) or without (for FNSTENV) checking for pending unmasked floating-point exceptions, then mask all floating-point exceptions
- FRSTOR: load FPU state from 94-byte or 108-byte memory area (depending on the processor operating mode); this might unmask an exception pending in the FPU status word, that will be generated upon the execution of the next “waiting” FPU instruction
- FSAVE/FNSAVE: store FPU state into 94-byte or 108-byte memory area (depending on the processor operating mode) after (for FSAVE) or without (for FNSAVE) checking for pending unmasked floating-point exceptions, then re-initialize the FPU
- FINCSTP: increment the TOP field in the FPU status register (the tag and data registers are not affected)
- FDECSTP: decrement the TOP field in the FPU status register (the tag and data registers are not affected)
- FFREE: sets tag for any ST(i) to empty
- FNOP: no operation
- FWAIT/WAIT: check for and handle pending unmasked floating-point exceptions

Note that all the “non-waiting” floating-point instructions fall into the category of FPU control instructions: FNINIT, FNSTENV, FNSAVE, FNSTSW, FNSTCW, and FNCLEX do not check for pending unmasked exceptions, but are identical in rest with their counterparts listed above. This means that a software handler will not be invoked prior to executing one of these instructions when it follows another floating-point instruction that raised an unmasked floating-point exception. Except for FNSTSW and FNSTCW, these instructions either clear the FPU status word, or mask floating-point exceptions, which means that the pending exceptions are lost. For FNSTSW and FNSTCW, a subsequent “waiting” floating-point instruction can handle any pending exceptions.

2.7 Examples

The examples that follow illustrate various aspects of floating-point computations in general, or of computations with FPU instructions in particular.

The associated code is written in C (see [3]) with IA-32 assembly language statements inlined, mostly for FPU instructions. The “mov” instruction as used below moves an integer quantity between memory and an integer register, or between integer registers. The assembly language uses size specifiers for transfers to and from memory, e.g. DWORD PTR for 32-bit quantities, and TBYTE PTR for 80-bit quantities.

The floating-point constants used in the examples that follow are specified in hexadecimal, in the format mandated by the IEEE Standard [1] (which allows specifying values that sometimes cannot be expressed in decimal). For example, the single precision value 0x00800001 has to be separated into a 1-bit sign (0), an 8-bit biased exponent (00000001), and a 24-bit significand (100...001). The unbiased exponent is $1 - 127 = -126$, so the numeric value of the floating-point number encoded as 0x00800001 is $+1.00...01 * 2^{-126}$.

Example 3: The effect of rounding errors on equality tests

The purpose of this example is to point out a limitation inherent to floating-point arithmetic computations. Even when the IEEE Standard [1] is followed strictly, rounding errors still occur, and accuracy might be lost. If the expected result for a floating-point expression “fexpr” is “res”, a test of the kind:

```
if (fexpr == res)
    printf ("SUCCESS\n");
else
    printf ("FAIL\n");
```

will fail in most cases. Instead, it is recommended to test whether the calculated and the expected results are within a certain small interval determined by the real value “eps”:

```
if (-eps < fexpr - res && fexpr - res < eps)
    printf ("SUCCESS\n");
else
    printf ("FAIL\n");
```

To illustrate this, consider \sqrt{x} to be the infinitely precise value of the square root of x , and $(\sqrt{x})_m$ the value of \sqrt{x} rounded to nearest to the destination precision. The code fragment below checks whether the equality

$$((\sqrt{x})_m * (\sqrt{x})_m)_m = x$$

holds for a few simple values of x (computation performed in single precision):

```
#include <stdio.h>
void main () {
    float x, y, z;
    char *px, *py;
    int i;
    unsigned short cw, *pcw; // control word and pointer to it
    pcw = &cw;
    // set control word
    cw = 0x003f; // round to nearest, 24 bits, floating-point exc. disabled
    // cw = 0x043f; // round down, 24 bits, floating-point exc. disabled
    // cw = 0x083f; // round up, 24 bits, floating-point exc. disabled
    // cw = 0x0c3f; // round to zero, 24 bits, floating-point exc. disabled
    __asm {
        mov eax, DWORD PTR pcw
        fldcw [eax]
    }
    for (i = 0 ; i < 11 ; i++) {
        x = (float)i; // x = 1.0, 2.0, ..., 10.0
        // compute y = sqrt (x)
        px = (char *)&x;
        py = (char *)&y;
        __asm {
            mov eax, DWORD PTR px
            fld DWORD PTR [eax]
```

```

    fsqrt
    mov eax, DWORD PTR py
    fstp DWORD PTR [eax]
}
z = y * y;
printf ("x = %f = 0x%x\n", x, *(int *)&x);
printf ("y = %f = 0x%x\n", y, *(int *)&y);
printf ("z = %f = 0x%x\n", z, *(int *)&z);
if (z == x)
    printf ("EQUAL\n\n");
else
    printf ("NOT EQUAL\n\n");
}
}

```

For values of x tested which are not perfect squares, the equality test between x and z succeeds for $x = 3.0, 5.0,$ and $10.0,$ and it fails for $x = 2.0, 6.0, 7.0,$ and 8.0 if the rounding mode is set to nearest. The test fails for all the values of x that are not perfect squares if the rounding mode is not rounding to nearest.

Example 4: Tininess detection

This example shows code sample for the test for tininess detection from Example 1.

```

#include <stdio.h>

void main () {
    float a, b, c; // single precision numbers (of size 4 bytes)
    unsigned int u; // unsigned integer (of size 4 bytes)
    char *pa, *pb, *pc; // pointers to single precision numbers
    unsigned short sw, *psw; // status word and pointer to it
    unsigned short cw, *pcw; // control word and pointer to it
    // will compute c = a * b
    psw = &sw;
    pcw = &cw;
    // clear and read status word, set control word
    cw = 0x033f; // round to nearest, 64 bits, fp exc.disabled
    // cw = 0x073f; // round down, 64 bits, fp exc.disabled
    // cw = 0x0b3f; // round up, 64 bits, fp exc.disabled
    // cw = 0x0f3f; // round to zero, 64 bits, fp exc. disabled
    __asm {
        fclex
        mov eax, DWORD PTR pcw
        fldcw [eax]
        mov eax, DWORD PTR psw
        fstsw [eax]
    }
    printf ("BEFORE COMPUTATION sw = %4.4x\n", sw);
    pa = (char *)&a; u = 0x00fffffe;
    a = *(float *)&u; // a = 1.11...10 * 2^-126
    pb = (char *)&b; u = 0x3f000001;
}

```

```

b = *(float *)&u; // b = 1.00...01 * 2^-1
pc = (char *)&c;
// compute c = a * b
__asm {
    mov eax, DWORD PTR pa;
    fld DWORD PTR [eax]; // push a on the FPU stack
    mov eax, DWORD PTR pb;
    fld DWORD PTR [eax]; // push b on the FPU stack
    fmulp st(1), st(0); // a * b in st(1), pop st(0)
    mov eax, DWORD PTR pc;
    fstp DWORD PTR [eax]; // c = a * b from FPU stack to memory, pop st(0)
    mov eax, DWORD PTR psw
    fstsw [eax]
}
printf ("AFTER COMPUTATION sw = %4.4x\n", sw);
printf ("c = %8.8x = %f\n", *(unsigned int *)&c, c);
}

```

The output for rounding to nearest or to positive infinity shows the inexact-result status flag set, and the result of $1.0 * 2^{-126}$ (the smallest single precision normal):

```

BEFORE COMPUTATION sw = 0000
AFTER COMPUTATION sw = 0220
c = 00800000 = 0.000000

```

The output for rounding to negative infinity or toward zero shows the inexact-result and underflow status flags set, and the result of $0.11...1 * 2^{-126}$ (the largest single precision denormal):

```

BEFORE COMPUTATION sw = 0000
AFTER COMPUTATION sw = 0030
c = 007fffff = 0.000000

```

Example 5: Pure IEEE computations using the FPU instructions

The next example illustrates a major difference between the x87 floating-point computation model, and that recommended in [1]. The IEEE Standard for binary floating-point computations recommends that all intermediate calculations be performed in an IEEE supported format. The FPU computations follow a better model, while still complying entirely with the IEEE Standard. Even if the target precision for the final result of a given computation is for example single precision, the intermediate results are kept on the floating-point stack, with a larger exponent range (15 bits versus 8 bits in this example), and often with a larger precision (determined by the precision control field in the FPU control word, which is usually set to 53 or 64 bits – more than the 24 bits in the IEEE single precision format). Thus the FPU computation model allows for better accuracy, and avoids unwarranted overflow, underflow, or precision loss situations due to intermediate computations.

On the other side, if it is necessary to generate underflows or overflows for intermediate calculations as if the intermediate results were in single or double precision format, floating-point operations on the FPU stack have to be combined with stores to and loads from memory. To illustrate this, consider the single precision computation $d = (a * b) / c$, where $a = 1.0 * 2^{115}$, $b = 1.0 * 2^{125}$, and $c = 1.0 * 2^{120}$. Computed in single precision, the intermediate result $a * b = 1.0 * 2^{240}$ causes an overflow exception.

With overflow and inexact exceptions masked, the default result is positive infinity. The final result will be positive infinity too. Unlike in this “pure” IEEE model, the computation carried out in a typical FPU environment would store the intermediate result $a * b = 1.0 * 2^{240}$ on the FPU stack (no overflow occurs as the exponent range is of 15 bits), would calculate correctly $d = (a * b) / c = 1.0 * 2^{120}$, and would store this value to memory. The code below illustrates the FPU stack model in the first computation, and the “pure” IEEE single precision model in the second computation (for this, every intermediate result has to be stored to memory in order to trigger all the possible exceptions mandated by the IEEE model).

Note that the technique described here will not always work for double precision computations (but it will for single precision computations). This is due to possible double rounding errors that can occur when the result is a double precision denormal number: the result computed on the FPU stack will be rounded to 64 bits, and an error can occur when the `fst` instruction converts this value to a denormal number. For single precision computations, the double rounding error cannot occur in this case (see Example 6 below for more on double rounding errors). The problem could be solved for double precision computations though, by scaling the operands appropriately so as to force the denormalization (if any) to occur on the FPU stack, while rounding to 53-bit precision.

```
#include <stdio.h>
void main () {
    float a, b, c, d; // single precision floating-point numbers
    unsigned int u; // unsigned integer (of size 4 bytes)
    char *pa, *pb, *pc, *pd; // pointers to single precision numbers
    unsigned short sw, *psw; // status word and pointer to it
    // will compute d = (a * b) / c
    psw = &sw;
    // clear and read status word; set rounding to nearest,
    // and 64-bit precision
    __asm {
        finit
        mov eax, DWORD PTR psw
        fstsw [eax]
    }
    printf ("BEFORE COMP. sw = %4.4x\n", sw);
    pa = (char *)&a; u = 0x79000000;
    a = *(float *)&u; // a = 1.0 * 2^115
    pb = (char *)&b; u = 0x7e000000;
    b = *(float *)&u; // b = 1.0 * 2^125
    pc = (char *)&c; u = 0x7b800000;
    c = *(float *)&u; // c = 1.0 * 2^120
    pd = (char *)&d;
    // compute d = (a * b) / c holding the intermediate result
    // a * b = 2^240 on the FPU stack
    __asm {
        mov eax, DWORD PTR pa;
        fld DWORD PTR [eax]; // push a on the FPU stack
        mov eax, DWORD PTR pb;
        fld DWORD PTR [eax]; // push b on the FPU stack
        fmulp st(1), st(0); // a * b = 2^240 in st(1), pop st(0)
```


Example 6: Double rounding errors

The next example illustrates the possibility of having double rounding errors in floating-point computations where a correctly rounded result might be expected. This might occur for example when the infinitely precise result R of a computation needs to be rounded to a given destination precision, but instead is rounded first to a precision that is higher or has a larger exponent range than that of the destination precision. For example, assuming that the exponents fit in the destination format range, if ‘rn53’ signifies rounding to nearest to 53 bits in the significand and ‘rn64’ to 64 bits, then the equality

$$((R)_{m64})_{m53} = (R)_{m53}$$

does not always hold because of the double rounding errors on the left-hand side. It was shown that when R is the result of a simple arithmetic computation on normal floating-point numbers - add, subtract, multiply, divide, and square root - a double rounding error will not occur if the higher precision format of the first rounding (64 bits above) has in the significand more than twice the number of bits of the destination precision significand (53 bits above).

The example below calculates the single precision result of $1.00\dots01 \cdot 2^{-126} * 1.00010\dots0 \cdot 2^{-1}$ (the significands have 24 bits) in two ways. First, the result is rounded to the destination precision (24-bit significand) and placed on the FPU stack (15-bit exponent range), followed by storing to memory (24-bit significand and 8-bit exponent). This causes a double rounding error to occur: the result $0.100010\dots0 \cdot 2^{-126}$ is too small by one ulp. The second way to calculate this is to round the result to double precision on the FPU stack (53-bit significand and 15-bit exponent range), followed by storing to memory (24-bit significand and 8-bit exponent). The double rounding error does not occur, and the correctly rounded to nearest result of $0.100010\dots01 \cdot 2^{-126}$ is obtained. Note that if targeting a result in double precision, a similar double rounding error cannot always be avoided, as the next higher precision allows for only 64 bits in the significand, which is not more than twice larger than the 53-bit target precision.

```
#include <stdio.h>
void main () {
    float a, b, c; // single precision floating-point numbers
    unsigned int u; // unsigned integer (of size 4 bytes)
    char *pa, *pb, *pc; // pointers to single precision numbers
    unsigned short sw, *psw; // status word and pointer to it
    unsigned short cw, *pcw; // control word and pointer to it
    // will compute c = a * b
    psw = &sw;
    pcw = &cw;
    // clear status flags, read status word, set control word
    cw = 0x003f; // round to nearest, 24 bits, fp exc. disabled
    __asm {
        fclex
        mov eax, DWORD PTR pcw
        fldcw [eax]
        mov eax, DWORD PTR psw
        fstsw [eax]
    }
    printf ("BEFORE FIRST COMP. sw = %4.4x\n", sw);
    pa = (char *)&a; u = 0x00800001;
    a = *(float *)&u; // a = 1.00...01 * 2^-126
```

```

pb = (char *)&b; u = 0x3f080000;
b = *(float *)&u; // b = 1.00010...0 * 2^-1
pc = (char *)&c;
c = 123.0; // initialize c to random value
// compute c = a * b with 24 bits of precision;
// result a * b with `unbounded' exponent on FPU stack
__asm {
    mov eax, DWORD PTR pa
    fld DWORD PTR [eax] // push a on the FPU stack
    mov eax, DWORD PTR pb
    fld DWORD PTR [eax] // push b on the FPU stack
    fmulp st(1), st(0) // a * b in st(1), pop st(0)
    mov eax, DWORD PTR pc
    fstp DWORD PTR [eax] // c = a * b from FPU stack to memory, pop st(0)
    // read status word
    mov eax, DWORD PTR psw
    fstsw [eax]
}
printf ("AFTER FIRST COMP. sw = %4.4x\n", sw);
printf ("AFTER FIRST COMP. c = %8.8x = %f\n", *(unsigned int *)&c, c);
    // c = 0.100010...00 * 2^-126
// clear status flags, read status word, set control word
cw = 0x023f; // round to nearest, 53 bits, fp exc. disabled
__asm {
    fclex
    mov eax, DWORD PTR pcw
    fldcw [eax]
    mov eax, DWORD PTR psw
    fstsw [eax]
}
printf ("BEFORE SECOND COMP. sw = %4.4x\n", sw);
// compute c = a * b with 53 bits of precision;
// result a * b with `unbounded' exponent on FPU stack
__asm {
    mov eax, DWORD PTR pa
    fld DWORD PTR [eax] // push a on the FPU stack
    mov eax, DWORD PTR pb
    fld DWORD PTR [eax] // push b on the FPU stack
    fmulp st(1), st(0) // a * b in st(1), pop st(0)
    mov eax, DWORD PTR pc
    fstp DWORD PTR [eax] // c = a * b from FPU stack to memory, pop st(0)
    // read status word
    mov eax, DWORD PTR psw
    fstsw [eax]
}
printf ("AFTER SECOND COMP. sw = %4.4x\n", sw);

```

```

printf ("AFTER SECOND COMP. c = %8.8x = %f\n", *(unsigned int *)&c, c);
// c = 0.100010...01 * 2^-126
}

```

The output is:

```

BEFORE FIRST COMP. sw = 0000
AFTER FIRST COMP. sw = 0030
AFTER FIRST COMP. c = 00440000 = 0.000000
BEFORE SECOND COMP. sw = 0000
AFTER SECOND COMP. sw = 0230
AFTER SECOND COMP. c = 00440001 = 0.000000

```

Example 7: Raising an unmasked divide-by-zero exception

The following example illustrates raising an unmasked divide-by-zero floating-point exception fault caused by a divide instruction (FDIVP, when dividing PI by 0.0). The synchronizing instruction is FSTP (it could be an FWAIT placed right after FDIVP).

The `__try/__except` construct allows handling unmasked floating-point exceptions. If an unmasked floating-point exception is raised by a statement in the “`__try`” block, the action taken depends on the value of (or returned by) the filter expression that is the argument of `__except ()`. If this is “`EXCEPTION_EXECUTE_HANDLER`”, the code following the “`__except ()`” is executed (see [3] for other options).

```

#include <stdio.h>
#include <excpt.h>
void main () {
    float f;
    unsigned short cw, *pcw; // control word and pointer to it
    pcw = &cw;
    // clear status flags, set control word
    cw = 0x033b; // round to nearest, 64 bits, zero-divide exceptions enabled
    __asm {
        fclex
        mov eax, DWORD PTR pcw
        fldcw [eax]
    }
    __try {
        printf ("TRY BLOCK BEFORE DIVIDE BY 0\n");
        __asm {
            fldpi // load 3.1415926... in ST(0)
            fldz // load 0.0 in ST(0); 3.1415... in ST(1)
            fdivp st(1), st(0)
            // divide ST(1) by ST(0), result in ST(1), pop
            fstp f // store ST(0) in memory and pop stack top
        }
        printf ("TRY BLOCK AFTER DIVIDE BY 0 \n");
    } __except(EXCEPTION_EXECUTE_HANDLER) {
        printf ("EXCEPT BLOCK\n");
    }
}

```

```

    }
}

```

The unmasked exception causes the software handler to be invoked (the except block below), and the output will be:

```

TRY BLOCK BEFORE DIVIDE BY 0
EXCEPT BLOCK

```

If the FSTP instruction is commented out, the exception is not reported, and the output will be:

```

TRY BLOCK BEFORE DIVIDE BY 0
TRY BLOCK AFTER DIVIDE BY 0

```

Example 8: Raising an unmasked overflow exception when the excepting instruction has the destination in a memory location

The next example illustrates raising an overflow floating-point exception trap when the destination of the excepting instruction is a memory location (for the single precision computation of $1.0 \cdot 2^{115} * 1.0 \cdot 2^{125}$):

```

include <stdio.h>
#include <excpt.h>
void main () {
    float a, b, c; // single precision floating-point numbers
    unsigned int u; // unsigned integer (of size 4 bytes)
    char *pa, *pb, *pc; // pointers to single precision numbers
    unsigned short t[5], *pt;
    unsigned short sw, *psw; // status word and pointer to it
    unsigned short cw, *pcw; // control word and pointer to it
    psw = &sw;
    pcw = &cw;
    // clear exception flags, read status word, // set control word
    cw = 0x0337; // round to nearest, 64 bits, // overflow exceptions enabled
    __asm {
        fclex
        mov eax, DWORD PTR pcw
        fldcw [eax]
        mov eax, DWORD PTR psw
        fstsw [eax]
    }
    printf ("BEFORE COMP. sw = %4.4x\n", sw);
    pa = (char *)&a; u = 0x79000000;
    a = *(float *)&u; // a = 1.0 * 2^115
    pb = (char *)&b; u = 0x7e000000;
    b = *(float *)&u; // b = 1.0 * 2^125
    pc = (char *)&c;
    c = 0.0;
    pt = t;
}

```

```

__try {
    printf ("TRY BLOCK BEFORE OVERFLOW\n");
    // compute c = a * b
    __asm {
        mov eax, DWORD PTR pa
        fld DWORD PTR [eax] // push a on the FPU stack
        mov eax, DWORD PTR pb
        fld DWORD PTR [eax] // push b on the FPU stack
        fmulp st(1), st(0) // a * b in st(1), pop st(0)
        // cause the overflow exception
        mov eax, DWORD PTR pc
        fstp DWORD PTR [eax] // c = a * b from FPU stack to memory, pop st(0)
        fwait // trigger floating-point exception if any
    }
    printf ("TRY BLOCK AFTER OVERFLOW\n");
} __except(EXCEPTION_EXECUTE_HANDLER) {
    printf ("EXCEPT BLOCK\n");
    // clear exception flags, read status word, // set control word
    cw = 0x033f; // round to nearest, 64 bits, // exceptions disabled
    __asm {
        mov eax, DWORD PTR psw
        fnstsw [eax]
        fnclex
        mov eax, DWORD PTR pcw
        fldcw [eax]
    }
    printf ("sw = %4.4x\n", sw); // sw=0xb888: B=1, TOP=111, ES=1, OE=1
    __asm {
        mov eax, DWORD PTR pt
        fstp TBYTE PTR [eax] // c = a * b from FPU stack to memory, pop st(0)
    }
    printf ("t = %4.4x%4.4x%4.4x%4.4x%4.4x\n", t[4],t[3],t[2],t[1],t[0]);
        // t = 2^240
    }
}

```

The output shows the FPU status word value (sw=0xb888), with B=1, TOP=111, ES=1, and OE=1. The result (0x40ef8000000000000000, the double-extended precision encoding for 2^{240}) is unchanged on the FPU stack:

```

BEFORE COMP. sw = 0000
TRY BLOCK BEFORE OVERFLOW
EXCEPT BLOCK
sw = b888
t = 40ef8000000000000000

```

Note that the instruction raising the overflow floating-point exception is FSTP (when trying to convert the value on the FPU stack to the 32-bit single precision memory format).

Example 9: Raising an unmasked overflow exception when the excepting instruction has an FPU stack register as destination

A second example of raising an overflow floating-point exception trap is for the case when the destination of the excepting instruction is a location on the FPU stack (for the double-extended precision computation of $1.0...01 \cdot 2^{16000} * 1.0...01 \cdot 2^{16000}$ using rounding toward positive infinity):

```
#include <stdio.h>
#include <float.h>
#include <excpt.h>
void main () {
    unsigned short a[5], b[5], c[5], *pa, *pb, *pc;
    unsigned short sw, *psw; // status word and pointer to it
    unsigned short cw, *pcw; // control word and pointer to it
    psw = &sw;
    pcw = &cw;
    // clear exception flags, read status word, // set control word
    cw = 0x0b37; // round up, 64 bits, overflow exc. enabled
    __asm {
        fclex
        mov eax, DWORD PTR pcw
        fldcw [eax]
        mov eax, DWORD PTR psw
        fstsw [eax]
    }
    printf ("BEFORE COMP. sw = %4.4x\n", sw);
    // a = 1.0 * 2^16000, b = 1.0 * 2^16000
    a[4] = 0x7e7f; a[3] = 0x8000; a[2] = 0x0000; a[1] = 0x0000; a[0] = 0x0001;
    b[4] = 0x7e7f; b[3] = 0x8000; b[2] = 0x0000; b[1] = 0x0000; b[0] = 0x0001;
    pa = a; pb = b; pc = c;
    __try {
        printf ("TRY BLOCK BEFORE OVERFLOW\n");
        // compute c = a * b
        __asm {
            mov eax, DWORD PTR pa
            fld TBYTE PTR [eax] // push a on the FPU stack
            mov eax, DWORD PTR pb
            fld TBYTE PTR [eax] // push b on the FPU stack
            fmulp st(1), st(0) // a * b in st(1), pop st(0)
            fwait // trigger floating-point exception if any
        }
        printf ("TRY BLOCK AFTER OVERFLOW\n");
    } __except(EXCEPTION_EXECUTE_HANDLER) {
        printf ("EXCEPT BLOCK\n");
        // clear exceptions, read status word, set control word
        cw = 0x0b3f; // round up, 64 bits, exceptions disabled
        __asm {
```

```

    mov eax, DWORD PTR psw
    fnstsw [eax]
    fnclex
    mov eax, DWORD PTR pcw
    fldcw [eax]
}
printf ("sw = %4.4x\n", sw); // sw=0xbaa8:
    // B=1, TOP=111, C1=1, ES=1, PE=1, OE=1
__asm {
    mov eax, DWORD PTR pc
    fstp TBYTE PTR [eax] // c = a * b from FPU stack to memory, pop st(0)
}
printf ("c = %4.4x%4.4x%4.4x%4.4x%4.4x\n", c[4],c[3],c[2],c[1],c[0]);
    // c = 2^32000 / 2^24576 = 2^7424 (biased exponent is 0x5cff)
}
}

```

The output:

```

BEFORE COMP.  sw = 0000
TRY BLOCK BEFORE OVERFLOW
EXCEPT BLOCK
sw = baa8
c = 5cff800000000000000003

```

shows that the result on the FPU stack has been scaled down by 2^{24576} (0x5cff8000000000000003 is the double-extended encoding for $2^{32000-24576} = 2^{7424}$). The FPU status word (sw = baa8) indicates B=1, TOP=111, C1=1, ES=1, PE=1, OE=1 (C1=1 means that the significand of the result is larger than that of the infinitely precise result).

Note that the instruction raising the overflow floating-point exception is FMUL (as the exponent of 32000 exceeds the 15-bit range of the double-extended precision format).

3 Streaming SIMD Extensions

The Intel Streaming SIMD Extensions (SSE) instructions, integer and floating-point, were designed to enhance performance of advanced media and communications applications. New registers, data types and instructions (not only floating-point) are combined into a SIMD execution model to accelerate performance of applications. The only basic floating-point data type used by the SSE instructions is the single precision data type, with the characteristics shown in Table 1 (this data type is identical to the single precision memory format of the FPU). This means that the values that can be represented are zero, denormal numbers, normalized finite numbers, infinity, and NaNs. The single precision data type is convenient for applications involving image processing, audio synthesis, speech recognition, telephony, 2D and 3D graphics, and others.

3.1 Streaming SIMD Extensions Architecture

The SSE floating-point instructions operate on a new set of eight 128-bit registers (Figure 3), each of which can be addressed directly (unlike the FPU register stack model, although FXCH circumvents some of this limitation). The new registers can hold data, but cannot be used to address memory (addressing is achieved with the existing IA-32 addressing modes).

XMM7
XMM6
XMM5
XMM4
XMM3
XMM2
XMM1
XMM0

Figure 3. SIMD Floating-Point Registers

Each SIMD register can hold data in the new format of packed 128 bits, representing four single precision floating-point numbers (X1, X2, X3, X4 in Figure 4; X1 occupies the least significant position).

127	96 95	65 64	32 31	0
X4		X3		X1

Figure 4. Packed single precision floating-point data type

In memory, the new data type is stored in 16 consecutive bytes, in little endian ordering, but accesses and data transfers between memory and registers, or between registers only can occur in 128-bit or 32-bit transfers. Memory operands have to be aligned on a 16-byte boundary, except for unaligned loads and stores.

The SSE floating-point instructions operate either on the least significant operand (or pair of operands) from the packed operands in their scalar version, or on all four operands (or subsets of operands) in their parallel or packed version. The results are placed in the corresponding positions in the destination register or memory location. In the scalar operation case, the three upper components (occupying bits 127-32) are copied from the first source operand.

3.2 SIMD Control and Status Register

A new 32-bit control and status register (Figure 5) is associated with the SSE instructions. Bits 31 through 16 (not shown in the figure) and bit 6 are reserved, and non-zero values should not be written to them. The interaction between the FPU instructions or state, and the SSE instructions or state is established by the semantics of the application using them. It has to be noted that every SSE or SSE2 instruction (section 4) that contains a reference to an MMX technology register will trigger a transition from floating-point to MMX technology operations. This is achieved by setting TOP=0 in the FPU Status Register, and the tag register to 0, indicating a full FPU stack.

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

Figure 5. Lower 16 bits of the control and status register MXCSR

Bits 15 through 7 control the flush-to-zero mode, the rounding mode, and masking/unmasking of floating-point exceptions. Bits 5 through 0 contain floating-point exception status flags. The MXCSR register does not contain a precision control field (PC) to determine the precision of the floating-point calculations performed, as single precision is statically associated with all of the SSE instructions that operate on floating-point numbers.

Bit 15 controls the flush-to-zero mode, and can be used to speed up applications that produce many tiny values. If the flush-to-zero mode is selected (by setting the FZ bit in MXCSR) and the underflow exceptions are masked, then any tiny result that would be produced by a floating-point instruction is replaced by the correctly signed zero, and the underflow and inexact status flags in MXCSR are set. For performance reasons, this is the recommended mode of running applications that take advantage of the SSE floating-point instructions. If the FZ bit is clear or the underflow exceptions are unmasked, then the flush-to-zero mode is not in effect.

Bits 14 and 13 represent the rounding control field (RC) that determines which IEEE rounding method [1] is used for results that cannot be exactly represented in the single precision destination format (round to nearest if RC=00B, down if RC=01B, up if RC=10B, and toward zero if RC=11B). Rounding of a result following a floating-point operation is performed as explained in Section 2 above.

Bits 12 through 7 (PM, UM, OM, ZM, DM, IM) are the exception flag masks. If set, they mask (disable) the SIMD floating-point exceptions (inexact-result, underflow, overflow, divide-by-zero, denormal operand, and invalid operation exceptions respectively).

Bits 5 through 0 (PE, UE, OE, ZE, DE, IE) are the exception flags for inexact-result, underflow, overflow, divide-by-zero, denormal operand, and invalid operation exceptions respectively, indicating that one or more floating-point exceptions have been detected since they were last cleared (they are “sticky” bits). Unlike the status flags in the FPU status word, the MXCSR status flags can be set as a result of the computation for any of up to four subsets of single precision operands of an SSE floating-point instruction. If the instruction operates on multiple sets of operands, the status flag value is the logical OR of the multiple hypothetical status flags associated with each subset.

Example 10: Effect of the flush-to-zero mode

The following computation illustrates the effect of the flush-to-zero mode. When using SSE floating-point instructions, pure single precision computations as specified by the IEEE Standard for Binary Floating-Point Arithmetic can be performed in one step. For example, MULSS is used for a scalar single precision multiply. This is not possible for FPU computations, where a pair of instructions – FMUL and FST – has to be used, the benefit being a potentially higher accuracy of the FMUL result, which is useful for intermediate computation steps. Consider once again the multiplication of the two single precision floating-point numbers a and b from Example 1, and assume that the underflow and inexact-result exceptions are masked, and the flush-to-zero mode is selected:

$$a = 1.1\dots10 \cdot 2^{-126} \approx 10^{-54.28}$$

$$b = 1.0\dots01 \cdot 2^{-1} \approx 0.5$$

The result of the hypothetical rounding to 24 bits in the significand and unbounded exponent depends on the rounding mode (the significands below have 24 bits):

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ for rounding to nearest (not tiny)}$$

$$a \cdot b = 1.11\dots1 \cdot 2^{-127} \text{ for rounding down (tiny)}$$

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ for rounding up (not tiny)}$$

$$a \cdot b = 1.11\dots1 \cdot 2^{-127} \text{ for rounding to zero (tiny)}$$

The actual rounded result (to 24 bits in the significand and bounded exponent) depends again on the rounding mode, but is also affected by the flush-to-zero mode (tiny results are flushed to the correctly signed zero), yielding the result below:

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ for rounding to nearest (set P flag)}$$

$$a \cdot b = +0.0 \text{ for rounding down (set P,U flags)}$$

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ for rounding up (set P flag)}$$

$$a \cdot b = +0.0 \text{ for rounding to zero (set P,U flags)}$$

3.3 Floating-Point Exceptions

The SIMD floating-point instructions may cause the same six types of floating-point exceptions as the FPU: invalid operation, denormal operand, divide-by-zero, overflow, underflow, and inexact-result (or precision). Each exception has associated with it a mask/unmask (disable/enable) bit in the MXCSR control/status word, and a “sticky” exception status flag.

Invalid operation, denormal operand and divide-by-zero exceptions are pre-computation exceptions (floating-point faults), while overflow, underflow, and inexact-result are post-computation exceptions (floating-point traps).

If an exception is detected that is masked (mask bit set), the SIMD FPU handles the exception automatically, producing a predefined result and allowing program execution to continue. Note that the processor may detect several floating-point exceptions while executing a single instruction, and thus the status flags in MXCSR are set as a logical-OR of the exception conditions for all the sub-operand computations.

If an exception is detected that is unmasked (mask bit clear), the SIMD FPU invokes a software exception handler to handle the exception, via the SIMD floating-point interrupt vector 19. This happens immediately, without waiting for another SIMD floating-point instruction to trigger the exception (as is the case for FPU floating-point exceptions). The processor leaves unchanged the source register operands prior to invoking the software handler. Similarly, the EFLAGS register will not be modified if COMISS or UCOMISS (described below) raises an unmasked exception. This is different from the x87 model for post-computation exceptions, where a result is provided to the software handler: no result is provided for exceptions raised by SSE floating-point instructions, regardless of whether the SIMD exception was a pre-computation or a post-computation one. The exception status flags in MXCSR will be updated by the logical-OR of the exception conditions for all the sub-operand computations (for post-computation unmasked exceptions, the flags will also be updated for any pre-computation exception conditions, all of which must have been masked).

The exception conditions are similar to those for the FPU, only that there are no unsupported operands in single precision format (so this will not be a possible cause for invalid operation exceptions).

The precedence of the SIMD floating-point exceptions is similar to that of the FPU exceptions:

- invalid operation exception (SNaN operand, or NaN for some floating-point compare instructions), or operand outside the admissible range
- QNaN operand (not an exception, but handling QNaN operands has priority over lower priority exceptions)
- invalid operation exceptions not mentioned above, or divide-by-zero exception
- denormal operand exception (if masked, execution continues and a lower priority exception can occur as well)
- numeric overflow or underflow exceptions, possibly in conjunction with the inexact-result exception
- inexact-result exception

Note that setting the value of a status flag in MXCSR and unmasking subsequently the corresponding floating-point exception will not cause an invocation of the software exception handler. This is unlike the similar situation for the FPU status and control words, where a floating-point exception would be raised in this case (triggered on the first “waiting” floating-point instruction to follow).

A second important difference between floating-point exceptions signaled by FPU instructions and those signaled by SSE floating-point instructions is that an unmasked exception for a SIMD instruction is reported immediately, rather than having to wait for a subsequent floating-point instruction to trigger it.

A third difference is that SSE floating-point load and store instructions do not signal numeric exceptions, unlike their FPU counterparts. This is partially due to the fact that data type conversions do not occur on SIMD loads and stores, which means that overflow, underflow, and inexact result exceptions are not possible.

Note that if a tiny result is flushed to zero (which means that the FZ bit and UM bits must have been set in MXCSR), and the inexact traps are unmasked (PM clear), a software exception handler will be invoked for the inexact-result exception. The result delivered to the exception handler will be the correctly signed zero.

As a final observation, note that some instructions (COMISS and UCOMISS, described below) write their result to the EFLAGS register. The value of EFLAGS is not updated in the presence of an unmasked floating-point exception (which can be an invalid operation or denormal operand exception).

3.4 Software Exception Handling

The typical operations performed by the software handler for an SSE floating-point instruction are similar to those for an FPU instruction. Clearing the MXCSR status flags is not necessary anymore, because unmasking a floating-point exception for which the status flag was previously set, will not cause the invocation of a software handler. The major complication in handling unmasked floating-point exceptions for SSE floating-point instructions that operate on packed operands is that the status flags in MXCSR do not indicate which of the four subsets of operands has caused the exception. In addition to this (unlike for x87 unmasked exceptions), a result is not provided in the destination SIMD register for floating-point traps (overflow, underflow, and inexact). To handle correctly the exception (and to comply with the recommendations of the IEEE Standard [1]), software has to emulate the excepting instruction piece by piece, and to identify which of the four components has raised the unmasked exception (note that there might be more than one unmasked exception raised by the same instruction). The corresponding subset of operands (and possibly a result if this is a floating-point trap) has to be presented to the user handler. If execution of the application containing the excepting instruction has to be continued, the result provided by the user handler has to be combined with the emulated results for the remaining subsets of non-excepting operands (if any), and execution of the interrupted application can be resumed.

3.5 Operating on NaNs

Similar to the case of the FPU instructions, QNaNs do not cause floating-point exceptions for most SSE floating-point instructions (except for the floating-point maximum/minimum instructions and some of the floating-point compare instructions). However, the rules for generating results are different from those for FPU instructions (presented in Table 2). Handling NaNs takes precedence over some floating-point exceptions, as seen above. Table 3 lists the rules for generating QNaN results for SSE instructions.

Table 3. Rules for Generating QNaN Results for SSE Instructions

Source Operands	QNaN Result
SNaN and QNaN	Src1 NaN, converted to QNaN if Src1 is SNaN
Two SNaNs	Src1 NaN, converted to QNaN
Two QNaNs	Src1 NaN
SNaN and a real value, or SNaN if only one operand	The SNaN, converted to a QNaN (quieted)
QNaN and a real value, or QNaN if only one operand	The QNaN source operand
No NaN operand, but an invalid operation exception is signaled	QNaN Real Indefinite

3.6 Streaming SIMD Extensions

The SSE extensions include mainly floating-point (arithmetic) instructions, but also instructions that control cacheability of all MMX technology and 32-bit IA-32 data types (not discussed here; see [2]). The instructions operating on all four components of their packed operands have the suffix “PS” (from “packed single precision”). The instructions operating only on the least significant components of their packed operands have the suffix “SS” (from “scalar single precision”). If a packed instruction reads its operand from memory, a 128-bit read is implied. If a scalar instruction reads its operand from memory, a 32-bit read is implied. The SSE instructions operating on floating-point numbers are enumerated below.

1. Data movement instructions:

- MOVAPS/MOVUPS: move aligned/unaligned packed single precision floating-point; transfers 128 bits of data between memory and SIMD floating-point registers, or between SIMD registers; floating-point exceptions: none
- MOVHPS/MOVLPS: move aligned, high/low packed single precision floating-point; transfers 64 bits of data between memory and the upper/lower half of a SIMD floating-point register (the lower/upper half is unchanged); floating-point exceptions: none
- MOVHLPS/MOVLHPS: move high/low to low/high packed single precision floating-point; transfers the upper/lower 64 bits of the source register into the lower/upper 64 bits of the destination register (the upper/lower 64 bits are left unchanged); floating-point exceptions: none
- MOVMSKPS: move mask packed, single precision floating-point to r32; transfers the most significant bit of the four packed operands to a 32-bit IA-32 integer register r32; floating-point exceptions: none
- MOVSS: move scalar single precision floating-point; transfers 32 bits between memory or the least significant 32 bits of a SIMD register, and the least significant 32 bits of a SIMD register; floating-point exceptions: none

2. Arithmetic instructions:

- ADDPS/ADDSS/SUBPS/SUBSS/MULPS/MULSS: add/subtract/multiply packed/scalar, single precision floating-point; first source operand and destination is in a SIMD register; second source operand is in a SIMD register or memory location; floating-point exceptions: I, D, O, U, P
- DIVPS/DIVSS: divide packed/scalar, single precision floating-point; first source operand and destination is in a SIMD register; second source operand is in a SIMD register or memory location; floating-point exceptions: I, Z, D, O, U, P
- SQRTPS/SQRTSS: square root packed/scalar, single precision floating-point; source operand is in a SIMD register or memory location; destination is a SIMD register; floating-point exceptions: I, D, P

3. Maximum and minimum instructions:

- MAXPS/MAXSS/MINPS/MINSS: maximum/minimum packed/scalar, single precision floating-point; first source operand and destination is in a SIMD register; second source operand is in a SIMD register or memory location; floating-point exceptions: I, D (the invalid operation exception is triggered by any NaN operand)

4. Comparison instructions:

- CMPPS/CMPSS: compare packed/scalar, single precision floating-point; first source operand and destination is in a SIMD register; second source operand is in a SIMD register or memory location; the result is a 32-bit mask of 1's (true) or 0's (false); floating-point exceptions: I, D (the invalid operation exception is triggered by any NaN operand for lt, le, nlt, nle comparisons, otherwise only by SNaN)
- COMISS/UCOMISS: compare scalar single precision floating-point ordered/unordered and set EFLAGS; first source operand and destination is the least significant component of a SIMD register; second source operand is the least significant component of a SIMD register, or a memory location; the result is written to the ZF, PF, and CF bits of the EFLAGS register; floating-point exceptions: I, D (the invalid operation exception is triggered by any NaN operand for COMISS, and by any SNaN operand for UCOMISS)

5. Conversion instructions:

- CVTPI2PS: convert two packed 32-bit signed integers from an MMX technology register or memory to two packed, single precision floating-point numbers in a SIMD register (in the two least significant positions); floating-point exceptions: P
- CVTSI2SS: convert one 32-bit signed integer from an integer register or memory to one single precision floating-point number in a SIMD register (in the least significant position); floating-point exceptions: P
- CVTTPS2PI/CVTTTPS2PI: convert the lower two packed single precision floating-point numbers in a SIMD register or two single precision floating-point numbers from memory to two packed 32-bit signed integers in an MMX technology register; the CVTTTPS2PI version uses the rounding to zero mode (truncate) instead of the one specified in MXCSR; floating-point exceptions: I, P
- CVTSS2SI/CVTTSS2SI: convert the single precision floating-point number in the least significant position in a SIMD register or a single precision floating-point number from memory to a 32-bit signed integer in an integer register; the CVTTSS2SI version uses the rounding to zero mode (truncate) instead of the one specified in MXCSR; floating-point exceptions: I, P

6. Logical instructions, bitwise:

- ANDPS/ANDNPS/ORPS/XORPS: packed logical AND, AND-NOT, OR, XOR; floating-point exceptions: none

7. Reciprocal approximation instructions:

- RCPPS/RCPSS: packed/scalar, single precision floating-point reciprocal approximation (with a relative error of at most $1.5 \cdot 2^{-12}$); source operand is in a SIMD register or memory location; destination is a SIMD register; floating-point exceptions: none
- RSQRTPS/RSQRTSS: packed/scalar, single precision floating-point square root reciprocal approximation (with a relative error of at most $1.5 \cdot 2^{-12}$); source operand is in a SIMD register or memory location; destination is a SIMD register; floating-point exceptions: none

8. State management instructions:

- FXSAVE/FXRSTOR: save/restore FP/MMX technology state and SIMD state, to/from a 512-byte area in memory: CS (code segment descriptor), IP (instruction pointer), FOP (floating-point operation code), FTW (FPU tag word), FSW (FPU status word), FCW (FPU control word), MXCSR (SIMD control/status word), DS (data segment descriptor), DP (data pointer), eight FPU stack/MMX technology registers, eight SIMD registers; floating-point exceptions: none
- STMXCSR/LDMXCSR: store/load SIMD control and status word to/from a 32-bit memory location; floating-point exceptions: none

FXSAVE and FXRSTOR are optimized, faster replacements for FSAVE/FRSTOR respectively (while saving/restoring more information).

Other SSE instructions operate on integer quantities. Many behave identically to original MMX instructions in the presence of x87 floating-point instructions. Other instructions allow shuffling 32-bit fields (corresponding to a single precision floating-point number) between SIMD registers or between a SIMD register and a memory location (but the destination is always a SIMD register). Finally, cacheability control instructions provide programmatic control for minimizing cache pollution when writing data to memory from either the MMX technology registers or the SIMD floating-point registers.

3.7 Writing Applications with SSE Instructions

The SSE floating-point instructions are accessible from all the IA-32 execution modes: Protected Mode, Real-Address Mode, and Virtual 8086 Mode. Before using them though, an application must check that the processor and the operating system support the SSE instructions:

- emulation disabled: CR0.EM(bit 2) = 0
- processor supports SSE: CPUID.XMM(EDX bit 25)=1
- processor supports FXSAVE/FXRSTOR: CPUID.FXSR(EDX bit 24)=1
- OS supports SIMD FP state during context switches: CR4.OSFXSR(bit 9)=1.

More checks are needed to verify support for additional non-floating-point SIMD instructions. (see [2] for more details). In addition, if SIMD floating-point exceptions are going to be unmasked, the application must check first that the operating system supports unmasked SSE exceptions:

- OS supports unmasked SIMD exceptions: CR4.OSXMMEXCPT(bit 10)=1

3.8 Examples

The two examples that follow illustrate simple computations with SSE instructions.

Example 11: Using the SSE Instructions

The first example performs the division of the SIMD operand (1.0, 1.0, 1.0, 1.0) by $(0.0\dots01 \cdot 2^{-126}, 0.0, 1.1\dots11 \cdot 2^{127}, \text{SNaN})$, with the flush-to-zero mode selected, the rounding mode set to nearest, and the floating-point exceptions masked. The first divide (from the left) will set the denormal operand status flag, and then the overflow status flag in MXCSR (the result is larger than the maximum single precision floating-point number). The second divide raises a divide-by-zero exception and sets the corresponding status flag. The third divide generates a tiny and inexact result, but the flush-to-zero mode that is selected replaces it by +0.0 and sets the underflow and precision status flags in MXCSR. The last divide

raises an invalid operation exception and sets the invalid status flag in MXCSR. The SIMD result will be (+inf, +inf, 0.0, QNaN), where QNaN is the quieted SNaN from the first input operand. The MXCSR will have all the status flags set, but without knowing the values of the input operands, it cannot be told which sub-operand caused which exception. If exceptions were unmasked, the software handler would have to emulate the execution of the instruction one component at a time in order to determine correctly the result.

```
#include <stdio.h>
void main () {
    char *mem;
    unsigned int uimem[4];
    int mxcsr, *pmxcsr;
    mem = (char *)uimem;
    // set and then read new value of MXCSR
    mxcsr = 0x00009f80;
        // ftz = 1, rc = 00 (to nearest), traps disabled, flags clear
    pmxcsr = &mxcsr;
    __asm {
        mov eax, DWORD PTR pmxcsr
        ldmxcsr [eax]
        stmxcsr [eax]
    }
    printf ("BEFORE SIMD DIVIDE: MXCSR = 0x%8.8x\n", mxcsr);
    // load first set of operands
    uimem[0] = 0x3f800000; // 1.0
    uimem[1] = 0x3f800000; // 1.0
    uimem[2] = 0x3f800000; // 1.0
    uimem[3] = 0x3f800000; // 1.0
    __asm {
        mov eax, DWORD PTR mem;
        movups XMM1, [eax];
    }
    // load second set of operands
    uimem[0] = 0x00000001; // 0.0...01 * 2-126
    uimem[1] = 0x00000000; // 0.0
    uimem[2] = 0x7f7fffffff; // 1.1...1 * 2127
    uimem[3] = 0x7fbf0000; // SNaN
    __asm {
        mov eax, DWORD PTR mem;
        movups XMM2, [eax];
    }
    // perform SIMD divide and store result to memory
    __asm {
        divps XMM1, XMM2;
        mov eax, DWORD PTR mem;
        movups [eax], XMM1;
    }
}
```

```

// read new value of MXCSR
__asm {
    mov eax, DWORD PTR pmxcsr
    stmxcsr [eax]
}
printf ("AFTER SIMD DIVIDE: MXCSR = 0x%8.8x\n", mxcsr);
printf ("res = %8.8x %8.8x %8.8x %8.8x = %f %f %f %f\n",
        uimem[0], uimem[1], uimem[2], uimem[3],
        *(float *)&uimem[0], *(float *)&uimem[1],
        *(float *)&uimem[2], *(float *)&uimem[3]);
}

```

The output is:

```

BEFORE SIMD DIVIDE: MXCSR = 0x00009f80
AFTER SIMD DIVIDE:  MXCSR = 0x00009fbf
Res = 7f800000 7f800000 00000000 7fff0000 =
      1.#INF00 1.#INF00 0.000000 1.#QNAN0

```

Note that having used MOVUPS to move SIMD operands between SIMD registers and memory, the memory operand did not have to be 16-byte aligned. The next example will use MOVAPS, which implies that memory operands have to be 16-byte aligned.

Example 12: SSE example for $1.0 / (\text{sqrt}(a) - 1.0)$

The second example illustrates the simple computation of $1.0 / (\text{sqrt}(a) - 1.0)$ for four values of a , including one that is very close to 1.0: $a = 1.0...01 \cdot 2^0 = 1 + 2^{-23}$. The exact result (which can be obtained with a Taylor series expansion) is in this case:

$$R = (1 + 2^{-25} - 2^{-50} + 2^{-74} - 2^{-97} + \dots) \cdot 2^{24}$$

The single precision operand $a = 1.0...01$ is contained in the least significant component of XMM1 at the beginning of the computation.

```

#include <stdio.h>
void main () {
    char *mem;
    unsigned int *uimem;
    mem = (char *)(((int)malloc (144) + 16) & ~0x0f); // 16-byte aligned
    uimem = (unsigned int *)mem;
    // load x[i] in XMM1, i = 0,3
    uimem[0] = 0x40000000; // 2.0
    uimem[1] = 0x40400000; // 3.0
    uimem[2] = 0x40800000; // 4.0
    uimem[3] = 0x3f800001; // 1.0 + 1 ulp (1.0 + 2^-23)
    __asm {
        mov eax, DWORD PTR mem;
        movaps XMM1, [eax];
    }
    // load y[i] = 1.0 in XMM2, i = 0,3
    uimem[0] = 0x3f800000; // 1.0
    uimem[1] = 0x3f800000; // 1.0
}

```

```

uimem[2] = 0x3f800000; // 1.0
uimem[3] = 0x3f800000; // 1.0
__asm {
    mov eax, DWORD PTR mem;
    movaps XMM2, [eax];
}
// calculate 1.0 / (sqrt (x[i]) - 1.0), i = 0,3
__asm {
    // calculate sqrt (x[i]) in XMM1, i = 0,3
    sqrtps XMM1, XMM1;
    // calculate sqrt (x[i]) - 1.0 in XMM1, i = 0,3
    subps XMM1, XMM2;
    // calculate 1.0 / (sqrt (x[i]) - 1.0) in XMM2, i = 0,3
    divps XMM2, XMM1;
    // store result in memory
    mov eax, DWORD PTR mem;
    movaps [eax], XMM2;
}
printf ("res = %8.8x %8.8x %8.8x %8.8x = %f %f %f %f\n",
        uimem[0], uimem[1], uimem[2], uimem[3],
        *(float *)&uimem[0], *(float *)&uimem[1],
        *(float *)&uimem[2], *(float *)&uimem[3]);
}

```

The output:

```

res = 401a827a 3faed9ec 3f800000 7f800000 =
      2.414214 1.366025 1.000000 1.#INF00

```

shows that overflow has occurred in the computation for $a = 1 + 2^{-23}$. In this case, the single precision range is not sufficient to obtain an accurate result (if this makes any difference for the application). The same computation will be performed again in a different example, using SSE2 and the double extended precision format.

4 Streaming SIMD Extensions 2 Instructions

The Intel Streaming SIMD Extensions 2 (SSE2) instructions (many operating on floating-point data) were designed to enhance performance of MMX technology and scientific applications over the earlier generations of IA processors. The programming model is similar to that of the MMX technology instructions and of the SSE instructions, except that instructions now operate on two new data types: 128-bit wide integer, and 128-bit wide floating-point containing two double precision values. New floating-point instructions are combined into a SIMD execution model to accelerate performance of applications. The only basic floating-point data type used is the double precision data type, with the characteristics shown in Table 1 (this data type is identical to the double precision memory format of the FPU). This means that the values that can be represented are zero, denormal numbers, normalized finite numbers, infinity, and NaNs. The double precision data type was chosen because it is convenient for most technology and scientific applications. If more precision or a wider range is needed, the double-extended precision format provided by the FPU instructions has to be used.

4.1 Streaming SIMD Extensions 2 Architecture

The SSE2 instructions for floating-point computations operate on the same set of eight 128-bit SIMD floating-point registers (XMM registers) as the SSE floating-point instructions (Figure 3), each of which can be directly addressed. The SSE2 state does not require any new OS support for saving and restoring during a context switch, beyond that provided for saving and restoring the SSE state.

Each SIMD register can hold data in the new format of packed 128 bits, representing two double precision floating-point numbers (X1, X2 in Figure 6; X1 occupies the least significant position).

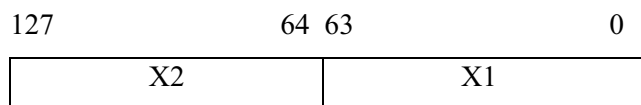


Figure 6. Packed double precision floating-point data type

In memory, the new data type is stored in 16 consecutive bytes, in little endian ordering, but accesses and data transfers between memory and registers, or between registers can only occur in 128-bit transfers or 64-bit transfers. Memory operands have to be aligned on a 16-byte boundary, except for unaligned loads and stores.

4.2 SIMD Control and Status Register

The floating-point control and status register (MXCSR) introduced for the SSE instructions is used also for the SSE2 instructions. The MXCSR register does not contain a precision control field (PC) to determine the precision of the floating-point calculations performed, as double precision is statically associated with the SSE2 instructions that operate on floating-point numbers.

The exception flags indicate that one or more floating-point exceptions have been detected since they were last cleared (they are “sticky” bits). Similar to the case of the SSE instructions, the MXCSR status flags can be set as a result of the computation for any of one or two subsets of double precision operands of an SSE2 floating-point arithmetic instruction. If the instruction operates on two sets of operands, the status flag value is the logical OR of the two hypothetical status flags associated with each subset.

The SSE2 floating-point instructions operate either on the least significant operand (or pair of operands) from the packed operands in their scalar version, or on both operands (or subsets of operands) in their parallel or packed version. The results are placed in the corresponding positions in the destination

register or memory location. In the scalar operation case, the upper component (occupying bits 64-127) is copied from the first source operand.

4.3 Floating-Point Exceptions

The SSE2 instructions may cause the same six types of floating-point exceptions as the FPU and the SSE instructions: invalid operation, denormal operand, divide-by-zero, overflow, underflow, and inexact-result (or precision). Each exception has associated with it a mask/unmask (disable/enable) bit in the MXCSR control/status word, and a “sticky” exception status flag bit (the MXCSR is used in the same way as for the SSE instructions).

Invalid operation, denormal operand and divide-by-zero exceptions are pre-computation exceptions (floating-point faults), while overflow, underflow, and inexact-result are post-computation exceptions (floating-point traps).

If an exception is detected that is masked (mask bit set), the SIMD FPU handles the exception automatically, producing a predefined result and allowing program execution to continue. Note that the processor may detect several floating-point exceptions while executing a single instruction, and thus the status flags in MXCSR are set as a logical-OR of exception conditions for all sub-operand computations.

If an exception is detected that is unmasked (mask bit clear), the SIMD FPU invokes a software exception handler to handle the exception, via the SIMD floating-point interrupt vector 19. This happens immediately, without waiting for another SIMD floating-point instruction to trigger the exception (as is the case for FPU floating-point exceptions). The processor leaves unchanged the source register operands prior to invoking the software handler. Similarly, the EFLAGS register will not be modified if COMISD or UCOMISD (described below) raises an unmasked exception. This is different from the FPU model for post-computation exceptions, where a result is provided to the software handler: no result is provided for exceptions raised by SSE2 instructions regardless of whether the SIMD exception was a pre-computation or a post-computation one. The exception status flags in MXCSR will be updated by the logical-OR of the exception conditions for all the sub-operand computations (for post-computation unmasked exceptions, the flags will also be updated for any pre-computation exception conditions, all of which must have been masked).

The exception conditions are similar to those for the FPU, only that there are no unsupported operands in double precision format (so this will not be a cause for invalid operation exceptions).

The precedence of the SSE2 floating-point exceptions is similar to that of the FPU exceptions, and the same as that presented above for SSE floating-point exceptions.

Note that setting the value of a status flag in MXCSR and unmasking subsequently the corresponding floating-point exception will not cause an invocation of the software exception handler. This is unlike the similar situation for the FPU status and control words, where a floating-point exception would be raised in this case, but similar to the SSE behavior.

A second important difference between floating-point exceptions signaled by FPU instructions and those signaled by SSE2 instructions (just as SSE floating-point instructions) is that an unmasked exception for a SIMD instruction is reported immediately, rather than having to wait for a subsequent floating-point instruction to trigger it.

A third difference is that the SSE2 instructions (like the SSE instructions) that load and store floating-point values do not signal numeric exceptions as their FPU counterparts. This is partially due to the fact that data type conversions do not occur on SIMD loads and stores, which means that overflow, underflow, and inexact exceptions are not possible.

Note that if a tiny result is flushed to zero (which means that the FZ bit and UM bits must have been set in MXCSR), and the inexact traps are unmasked (PM clear), a software exception handler will be invoked for the inexact-result exception. The result delivered to the exception handler will be the correctly signed zero.

As a final observation, note that some instructions (COMISD and UCOMISD, described below) write their result to the EFLAGS register. The value of EFLAGS is not updated in the presence of an unmasked floating-point exception (which can be an invalid operation or denormal operand exception).

4.4 Software Exception Handling

The typical operations performed by the software handler for an SSE2 instruction are similar to those for FPU or SSE instructions. Clearing the MXCSR status flags is not necessary anymore, because unmasking a floating-point exception for which the status flag was previously set, will not cause the invocation of a software handler. The major complication in handling unmasked SSE2 floating-point exceptions that operate on packed operands (just as the SSE instructions) is that the status flags in MXCSR do not indicate which of the two subsets of operands has caused the exception. In addition to this (unlike for x87 unmasked exceptions), a result is not provided in the destination SIMD register for floating-point traps (overflow, underflow, and inexact). To handle correctly the exception (and to comply with the recommendations of the IEEE Standard [1]), software has to emulate the excepting instruction piece by piece, and to identify which of the two components has raised the unmasked exception. The corresponding subset of operands (and possibly a result if this is a floating-point trap) have to be presented to the user handler. If execution of the application containing the excepting instruction has to be continued, the result provided by the user handler is combined with the emulated result for the remaining subset of non-excepting operands (if any), and execution of the interrupted application is resumed.

4.5 Operating on NaNs

Similar to the case of the FPU and SSE exceptions, QNaNs do not cause floating-point exceptions for most SSE2 that operate on floating-point values (exceptions are the floating-point maximum/minimum instructions and some of the floating-point compare instructions). The rules that apply here are different from the rules for FPU instructions (presented in Table 2 above), but the same as those from the SSE exceptions. Table 3, that lists the rules for generating QNaN results for SSE exceptions, applies also for SSE2 exceptions. Handling NaNs takes precedence over some floating-point exceptions, as seen above.

4.6 SSE2 Instructions

SSE2 instructions include mainly floating-point (arithmetic) instructions, but also instructions that control cacheability of all MMX technology and 32-bit IA-32 data types (not discussed here; see [2]). The instructions operating on both components of their packed operands have the suffix “PD” (from “packed double precision”). The instructions operating only on the least significant components of their packed operands have the suffix “SD” (from “scalar double precision”). If a packed instruction reads its operand from memory, a 128-bit read is implied. If a scalar instruction reads its operand from memory, a 64-bit read is implied. The SSE2 instructions operating on floating-point numbers are enumerated below.

1. Data movement instructions:

- MOVAPD/MOVUPD: move aligned/unaligned packed double precision floating-point; transfers 128 bits of data between memory and SIMD floating-point registers, or between SIMD registers; floating-point exceptions: none
- MOVHPD/MOVLDP: move aligned, high/low packed double precision floating-point; transfers 64 bits of data between memory and the upper/lower half of a SIMD floating-point register (the lower/upper half in unchanged); floating-point exceptions: none
- MOVMSKPD: move mask packed, double precision floating-point to r32; transfers the most significant bit of the two packed operands to a 32-bit IA-32 integer register r32; floating-point exceptions: none
- MOVSD: move scalar double precision floating-point; transfers 64 bits between memory or the least significant 64 bits of a SIMD register and the least significant 64 bits of a SIMD register; floating-point exceptions: none

2. Arithmetic instructions:

- ADDPD/ADDSD/SUBPD/SUBSD/MULPD/MULSD: add/subtract/multiply packed/scalar, double precision floating-point; first source operand and destination is in a SIMD register; second source operand is in a SIMD register or memory location; floating-point exceptions: I, D, O, U, P
- DIVPD/DIVSD: divide packed/scalar, double precision floating-point; first source operand and destination is in a SIMD register; second source operand is in a SIMD register or memory location; floating-point exceptions: I, Z, D, O, U, P
- SQRTPD/SQRTSD: square root packed/scalar, double precision floating-point; source operand is in a SIMD register or memory location; destination is a SIMD register; floating-point exceptions: I, D, P

3. Maximum and minimum instructions:

- MAXPD/MAXSD/MINPD/MINSD: maximum/minimum packed/scalar, double precision floating-point; first source operand and destination is in a SIMD register; second source operand is in a SIMD register or memory location; floating-point exceptions: I, D (the invalid operation exception is triggered by any NaN operand)

4. Comparison instructions:

- CMPPD/CMPSD: compare packed/scalar, double precision floating-point; first source operand and destination is in a SIMD register; second source operand is in a SIMD register or memory location; the result is a 64-bit mask of 1's (true) or 0's (false); floating-point exceptions: I, D (the invalid operation exception is triggered by any NaN operand for lt, le, nlt, nle comparisons, otherwise only by SNaN)
- COMISD/UCOMISD: compare scalar double precision floating-point ordered/unordered and set EFLAGS; first source operand and destination is the low half of a SIMD register; second source operand is the low half of a SIMD register, or a memory location; the result is written to the ZF, PF, and CF bits of the EFLAGS register; floating-point exceptions: I, D (the invalid

operation exception is triggered by any NaN operand for COMISD, and by any SNaN operand for UCOMISD)

5. Conversion instructions

- CVTPD2PI: convert packed double precision floating-point numbers from SIMD register or from memory to packed 32-bit signed integers in an MMX technology register, using the rounding mode specified by MXCSR
- CVTSD2SI: convert one double precision floating-point number from the lower half of a SIMD register or from memory to a 32-bit signed integer in a 32-bit IA-32 integer register, using the rounding mode specified by MXCSR
- CVTTPD2PI: convert packed double precision floating-point numbers from SIMD register or from memory to packed 32-bit signed integers in MMX technology register, using rounding to zero
- CVTTSD2SI: convert one double precision floating-point number from lower half of a SIMD register or from memory to a 32-bit signed integer in a 32-bit IA-32 integer register, using rounding to zero
- CVTPI2PD: convert lowest two 32-bit signed integers from an MMX technology register or from memory, to two double precision floating-point numbers in a SIMD register
- CVTSI2SD: convert one 32-bit signed integer from a 32-bit IA-32 integer register or from memory, to a double precision floating-point number in the lower half of a SIMD register
- CVTPD2DQ/CVTTPD2DQ: convert two double precision floating-point numbers from a SIMD register or from memory, to two 32-bit integers in the lower half of a SIMD register, using the rounding mode specified by MXCSR for CVTPD2DQ, or rounding to zero for CVTTPD2DQ
- CVTDQ2PD: convert two 32-bit signed integers from the lower half of a SIMD register or from memory to two packed double precision floating-point numbers in a SIMD register
- CVTPS2PD: convert two single precision floating-point numbers from the lower half of a SIMD register or from memory, to two double precision floating-point numbers in a SIMD register
- CVTSS2SD: convert a single precision floating-point number from the least significant position in a SIMD register or from memory, to a double precision floating-point number in the lower half of a SIMD register
- CVTPD2PS: convert two double precision floating-point numbers from a SIMD register or from memory, to two single precision floating-point numbers in the lower half of a SIMD register
- CVTSD2SS: convert a double precision floating-point number from the lower half of a SIMD register or from memory, to a single precision floating-point numbers in the least significant position of a SIMD register
- CVTPS2DQ/CVTTPS2DQ: convert four single precision floating-point numbers from a SIMD register or from memory, to four 32-bit signed integers in a SIMD register, using the rounding mode specified by MXCSR for CVTPS2DQ, or rounding to zero for CVTTPS2DQ
- CVTDQ2PS: convert four 32-bit signed integers from a SIMD register or from memory, to four single precision floating-point numbers in a SIMD register

6. Logical instructions, bitwise:
 - ANDPD/ANDNPD/ORPD/XORPD: packed logical AND, AND-NOT, OR, XOR; floating-point exceptions: none
7. State management instructions for the SSE2 instructions: these are the same as the state management instructions defined for the SSE: FXSAVE, FXRSTOR, STMXCSR, LDMXCSR.

Other SSE2 instructions operate on integer quantities, or allow shuffling 64-bit fields (corresponding to a double precision floating-point number) between SIMD registers or between a SIMD register and a memory location (but the destination is always a SIMD register). Finally, cacheability control instructions are the same as those defined for the SSE instructions.

4.7 Writing Applications with SSE2 Extensions

SSE2 extensions are accessible from all the IA-32 execution modes: Protected Mode, Real-Address Mode, and Virtual 8086 Mode. Before using them, an application must check that the processor and the operating system support the SSE2 extensions:

- emulation disabled: CR0.EM(bit 2) = 0
- processor supports SSE2: CPUID.WNI=1
- processor supports FXSAVE/FXRSTOR: CPUID.FXSR(EDX bit 24)=1
- OS supports SIMD FP state during context switches: CR4.OSFXSR(bit 9)=1.

More checks are needed to verify support for additional non-floating-point SIMD instructions (see [2] for more details). In addition, if SIMD floating-point exceptions are going to be unmasked, the application must check first that the operating system supports unmasked SSE2 exceptions:

- OS supports unmasked SIMD exceptions: CR4.OSXMMEXCPT(bit 10)=1

4.8 Example

Example 13: SSE2 example for $1.0 / (\sqrt{a} - 1.0)$

This example presents the double precision variant of the simple computation from Example 12 for SSE: $1.0 / (\sqrt{a} - 1.0)$ for two values of a , including one that is very close to 1.0: $a = 1.0...010...0 \cdot 2^0 = 1 + 2^{-23}$. The exact result is:

$$R = (1 + 2^{-25} - 2^{-50} + 2^{-74} - 2^{-97} + \dots) \cdot 2^{24}$$

and will be used to evaluate the relative error of the result. The double precision operand $a = 1.0...010...0$ is contained in the least significant half of XMM1 at the beginning of the computation.

```
#include <stdio.h>
void main () {
    char *mem;
    unsigned int *uimem;
    mem = (char *)(((int)malloc (144) + 16) & ~0x0f); // 16-byte aligned
    // printf ("mem = %x\n\n", (int)mem);
    uimem = (unsigned int *)mem;
    // load x[i] in XMM1, i = 0,1
```

```

uimem[1] = 0x40000000; uimem[0] = 0x00000000;
    // 2.0 (in uimem[1], uimem[0])
uimem[3] = 0x3ff00000; uimem[2] = 0x20000000;
    // 1.0 + 2^-23 (in uimem[3], uimem[2])
__asm {
    mov eax, DWORD PTR mem;
    movaps XMM1, [eax];
}
// load y[i] = 1.0 in XMM2, i = 0,1
uimem[1] = 0x3ff00000; uimem[0] = 0x00000000; // 1.0
uimem[3] = 0x3ff00000; uimem[2] = 0x00000000; // 1.0
__asm {
    mov eax, DWORD PTR mem;
    movaps XMM2, [eax];
}
// calculate 1.0 / (sqrt (x[i]) - 1.0), i = 0,1
__asm {
    // calculate sqrt (x[i]) in XMM1, i = 0,1
    sqrtpd XMM1, XMM1;
    // calculate sqrt (x[i]) - 1.0 in XMM1, i = 0,1
    subpd XMM1, XMM2;
    // calculate 1.0 / (sqrt (x[i]) - 1.0) in XMM2, i = 0,1
    divpd XMM2, XMM1;
    // store result in memory
    mov eax, DWORD PTR mem;
    movaps [eax], XMM2;
}
printf ("res = %8.8x%8.8x %8.8x%8.8x = %f %f\n",
        uimem[1], uimem[0], uimem[3], uimem[2],
        *(double *)&uimem[0], *(double *)&uimem[2]);
}

```

The output:

```
res = 4003504f333f9de5 4170000008000004 = 2.414214 16777216.500000
```

shows that for $a = 1 + 2^{-23}$ (from `uimem[3]`, `uimem[2]`), the computed value of R is $R^* = (1 + 2^{-25} + 2^{-50}) \cdot 2^{24}$. The relative error is

$$\varepsilon = |(R - R^*) / R| = (2^{-49} - 2^{-74} + 2^{-97} - \dots) / (1 + 2^{-25} - 2^{-50} + 2^{-74} - 2^{-97} + \dots) \approx 2^{-49}$$

which is much better than for the similar single precision calculation from Example 12 (where the computed result was positive infinity). A similar double-extended precision calculation (not shown here) would yield a relative error about 1.6 times smaller ($\varepsilon \approx 5 \cdot 2^{-52}$).

5 Summary of Differences

The main differences that occur when performing floating-point computations with IA-32 FPU instructions, SSE instructions, and SSE2 instructions are listed in Table 4 below.

Table 4. Differences when computing with IA-32 FPU instructions, SSE instructions, and SSE2 instructions

FPU Instructions	SSE Instructions	SSE2 Instructions
FPU instructions can be used at any time	SSE instructions can be used only after checking for processor and OS support	SSE2 instructions can be used only after checking for processor and OS support
FPU exceptions can be unmasked at any time	If the processor and the OS support SSE exceptions, floating-point exceptions can be unmasked only after checking for OS support	If the processor and the OS support SSE2 exceptions, floating-point exceptions can be unmasked only after checking for OS support
Scalar instructions	4-way SIMD instructions	2-way SIMD instructions
Floating-point formats: single, double, IA-32 stack single, IA-32 stack double and double-extended precision	Floating-point formats: single precision	Floating-point formats: double precision
Unsupported operands are possible (and will cause invalid operation floating-point exceptions)	Unsupported operands are not possible	Unsupported operands are not possible
Eight 80-bit stacked register set	Eight 128-bit linear register set (same as for SSE2)	Eight 128-bit linear register set (same as for SSE)
Unique load/store instructions for both aligned and unaligned memory accesses	Separate load/store instructions for aligned and unaligned memory accesses	Separate load/store instructions for aligned and unaligned memory accesses
Stack overflow or underflow exceptions are possible	No stack model	No stack model
Separate FPU control word and status word	Combined control/status word, MXCSR (same as for SSE2)	Combined control/status word, MXCSR (same as for SSE)

continued

Table 4. Differences when computing with IA-32 FPU instructions, SSE instructions, and SSE2 instructions (continued)

FPU Instructions	SSE Instructions	SSE2 Instructions
No flush-to-zero mode	Flush-to-zero mode available	Flush-to-zero mode available
Status flags identify uniquely the operation that has set them	Status flags do not always identify uniquely the operation that has set them – they are the logical OR of the results of up to four sub-operations	Status flags do not always identify uniquely the operation that has set them – they are the logical OR of the results of up to two sub-operations
Use interrupt vector 2 or 16 for software floating-point exception handler	Use interrupt vector 19 for software floating-point exception handler	Use interrupt vector 19 for software floating-point exception handler
Floating-point load and store may signal floating-point exceptions	Floating-point load and store do not signal floating-point exceptions	Floating-point load and store do not signal floating-point exceptions
Unmasked floating-point exceptions are deferred, and triggered only on the next “waiting” floating-point instruction	Unmasked floating-point exceptions are reported immediately, on the instruction that caused them	Unmasked floating-point exceptions are reported immediately, on the instruction that caused them
Input operands are provided to the exception handler for unmasked faults (I, D, Z)	Input operands are provided to the exception handler for unmasked faults (I, D, Z); emulation is necessary in order to handle the non-excepting sub-operations of packed operations	Input operands are provided to the exception handler for unmasked faults (I, D, Z); emulation is necessary in order to handle the non-excepting sub-operations of packed operations
Results (possibly scaled) are provided to the exception handler for unmasked traps (O, U, P)	Only input operands are provided to the exception handler for unmasked traps (O, U, P); software emulation is required in order to calculate the result for each sub-operation	Only input operands are provided to the exception handler for unmasked traps (O, U, P); software emulation is required in order to calculate the result for each sub-operation
Setting a status flag and then unmasking the corresponding floating-point exception triggers the exception	Setting a status flag and then unmasking the corresponding floating-point exception does not cause the exception to be raised	Setting a status flag and then unmasking the corresponding floating-point exception does not cause the exception to be raised

continued

Table 4. Differences when computing with IA-32 FPU instructions, SSE instructions, and SSE2 instructions (continued)

FPU Instructions	SSE Instructions	SSE2 Instructions
Transcendental functions are provided in the FPU instruction set	No transcendental functions are provided	No transcendental functions are provided
The hardware complies 100% with the mandatory requirements of the IEEE Std 754-1985; complies with all recommendations, except for allowing larger precision operands to generate smaller precision results (adding the benefit of higher accuracy)	The hardware complies 100% with the mandatory requirements of the IEEE Std 754-1985 for single precision computations; requires software emulation to implement unmasked floating-point exception handling (a standard recommendation)	The hardware complies 100% with the mandatory requirements of the IEEE Std 754-1985 for double precision computations; requires software emulation to implement unmasked floating-point exception handling (a standard recommendation)
FPU instructions treat NaN inputs differently (in some cases) from SSE and SSE2	SSE and SSE2 treat NaN inputs in the same way, but differently (in some cases) from FPU instructions	SSE2 and SSE treat NaN inputs in the same way, but differently (in some cases) from FPU instructions

The last example illustrates the differences in accuracy that can be expected when performing floating-point computations using FPU, SSE, or SSE2 instructions.

Example 14. Comparing the accuracy of computations using the FPU, SSE, and SSE2 Instructions

A simple expression is calculated, that has an exact result (which can be represented in any floating-point format):

$$(((1 / ((1 / 10) / (1 / 3))) + 3 / 10) / 11) * (1 / (1 / 99) + 11)) * 39 = 1417$$

The expression on the left-hand side is computed first with SSE instructions (packed operands are used, and the computation is actually replicated four times):

```
#include <stdio.h>
void main () {
    float res[4], *pres = res,
          a1[4] = {1.0, 1.0, 1.0, 1.0}, *pa1 = a1,
          a3[4] = {3.0, 3.0, 3.0, 3.0}, *pa3 = a3,
          a10[4] = {10.0, 10.0, 10.0, 10.0}, *pa10 = a10,
          a11[4] = {11.0, 11.0, 11.0, 11.0}, *pa11 = a11,
          a39[4] = {39.0, 39.0, 39.0, 39.0}, *pa39 = a39,
          a99[4] = {99.0, 99.0, 99.0, 99.0}, *pa99 = a99;
    __asm {
        mov eax, DWORD PTR pa1
        movups XMM5, [eax] // 1 in xmm5
        movaps XMM1, XMM5 // 1 in xmm1
        mov eax, DWORD PTR pa10
```

```

movups XMM2, [eax] // 10 in xmm2
divps XMM1, XMM2 // 1/10 in xmm1
movaps XMM2, XMM5 // 1 in xmm2
mov eax, DWORD PTR pa3
movups XMM3, [eax] // 3 in xmm3
divps XMM2, XMM3 // 1/3 in xmm2
divps XMM1, XMM2 // 3/10 in xmm1
movaps XMM2, XMM5 // 1 in xmm2
divps XMM2, XMM1 // 10/3 in xmm2
mov eax, DWORD PTR pa10
movups XMM1, [eax] // 10 in xmm1
divps XMM3, XMM1 // 3/10 in xmm3
addps XMM2, XMM3 // 109/30 in xmm2
mov eax, DWORD PTR pa11
movups XMM1, [eax] // 11 in xmm1
divps XMM2, XMM1 // 109/330 in xmm2
mov eax, DWORD PTR pa99
movups XMM3, [eax] // 99 in xmm3
movups XMM4, XMM5 // 1 in xmm4
divps XMM4, XMM3 // 1/99 in xmm4
divps XMM5, XMM4 // 99 in xmm5
addps XMM1, XMM5 // 110 in xmm1
mulps XMM1, XMM2 // 109/3 in xmm1
mov eax, DWORD PTR pa39
movups XMM2, [eax] // 39 in xmm2
mulps XMM1, XMM2 // 1417 in xmm1
mov eax, DWORD PTR pres;
movups [eax], XMM1;
}
printf ("res = \n\t%8.8x %8.8x %8.8x %8.8x = \n\t%f %f %f %f\n",
        *(unsigned int *)&res[0], *(unsigned int *)&res[1],
        *(unsigned int *)&res[2], *(unsigned int *)&res[3],
        res[0], res[1], res[2], res[3]);
}

```

The output corresponds to a pure IEEE single precision computation:

```

res = 44b12001 44b12001 44b12001 44b12001 =
      1417.000122 1417.000122 1417.000122 1417.000122

```

This shows that the result is larger than 1417 by 1 ulp:

$$\text{res} = 1417 + 1 \text{ ulp} = 1417 + 2^{10-23} = 1417 + 2^{-13}$$

The relative error corresponding to the absolute error $e = +2^{-13}$ is

$$\varepsilon_1 = 8.6147 \cdot 10^{-8}$$

The same result is obtained if the computation is carried out with FPU instructions on the FPU stack, with the precision set to 24 bits in the control word. Intermediate results do not have to be stored to memory and loaded back in this simple case in order to emulate the pure IEEE single precision model, because no overflow or underflow situations occur.

The same expression (that should have the result of 1417.0) is computed next with SSE2 instructions (packed operands are used, and the computation is replicated twice):

```
#include <stdio.h>
void main () {
    double res[2], *pres = res,
        a1[2] = {1.0, 1.0}, *pa1 = a1,
        a3[2] = {3.0, 3.0}, *pa3 = a3,
        a10[2] = {10.0, 10.0}, *pa10 = a10,
        a11[2] = {11.0, 11.0}, *pa11 = a11,
        a39[2] = {39.0, 39.0}, *pa39 = a39,
        a99[2] = {99.0, 99.0}, *pa99 = a99;
    unsigned int *uint;
    uint = (unsigned int *)res;
    __asm {
        mov eax, DWORD PTR pa1
        movupd XMM5, [eax] // 1 in xmm5
        movapd XMM1, XMM5 // 1 in xmm1
        mov eax, DWORD PTR pa10
        movupd XMM2, [eax] // 10 in xmm2
        divpd XMM1, XMM2 // 1/10 in xmm1
        movapd XMM2, XMM5 // 1 in xmm2
        mov eax, DWORD PTR pa3
        movupd XMM3, [eax] // 3 in xmm3
        divpd XMM2, XMM3 // 1/3 in xmm2
        divpd XMM1, XMM2 // 3/10 in xmm1
        movapd XMM2, XMM5 // 1 in xmm2
        divpd XMM2, XMM1 // 10/3 in xmm2
        mov eax, DWORD PTR pa10
        movupd XMM1, [eax] // 10 in xmm1
        divpd XMM3, XMM1 // 3/10 in xmm3
        addpd XMM2, XMM3 // 109/30 in xmm2
        mov eax, DWORD PTR pa11
        movupd XMM1, [eax] // 11 in xmm1
        divpd XMM2, XMM1 // 109/330 in xmm2
        mov eax, DWORD PTR pa99
        movupd XMM3, [eax] // 99 in xmm3
        movupd XMM4, XMM5 // 1 in xmm4
        divpd XMM4, XMM3 // 1/99 in xmm4
        divpd XMM5, XMM4 // 99 in xmm5
        addpd XMM1, XMM5 // 110 in xmm1
        mulpd XMM1, XMM2 // 109/3 in xmm1
        mov eax, DWORD PTR pa39
        movupd XMM2, [eax] // 39 in xmm2
        mulpd XMM1, XMM2 // 1417 in xmm1
        mov eax, DWORD PTR pres;
        movupd [eax], XMM1;
    }
```

```

}
printf ("res = \n\t%8.8x%8.8x %8.8x%8.8x = \n\t%f %f\n",
        uint[3], uint[2], uint[1], uint[0], res[1], res[0]);
}

```

The output corresponds to a pure IEEE double precision computation:

```
res = 409623ffffffffffe 409623ffffffffffe = 1417.000000 1417.000000
```

This shows that the result is 2 ulp-s less than 1417:

$$\text{res} = 1417 - 2 \text{ ulp} = 1417 - 2 \cdot 2^{10-52} = 1417 - 2^{-41}$$

The relative error corresponding to the absolute error $e = -2^{-41}$ is

$$\varepsilon_2 = 3.2092 \cdot 10^{-16}$$

This is much better than for the single precision, where the relative error was $\varepsilon_1 = 8.6147 \cdot 10^{-8}$.

The same result is obtained if the computation is carried out with FPU instructions on the FPU stack, with the precision set to 53 bits in the control word. Again, intermediate results do not have to be stored to memory and loaded back in this simple case in order to emulate the pure IEEE double precision model, because no overflow or underflow situations occur.

Finally, the expression considered is computed with FPU instructions, using the double-extended precision format for intermediate computations (PC=11 in the FPU Control Word):

```

#include <stdio.h>
void
main () {
    float a3 = 3., a10 = 10., a11 = 11., a39 = 39., a99 = 99.;
    char *pa3, *pa10, *pa11, *pa39, *pa99;
        // pointers to single precision numbers
    unsigned short t[5], *pt; // 10-byte (80-bit) result
    unsigned short cw, *pcw; // control word and pointer to it
    float res; // result, used just to print the decimal value
    char *pres;
    pa3 = (char *)&a3;
    pa10 = (char *)&a10;
    pa11 = (char *)&a11;
    pa39 = (char *)&a39;
    pa99 = (char *)&a99;
    pt = t;
    pres = (char *)&res;
    pcw = &cw;
    // set control word
    cw = 0x033f; // round to nearest, 64 bits, exceptions disabled
        // (double-extended precision)
    // cw = 0x023f; // (use for pure IEEE double precision)
        // round to nearest, 53 bits, exceptions disabled
    // cw = 0x003f; // (use for pure IEEE single precision)
        // round to nearest, 24 bits, exceptions disabled

```

```

__asm {
    mov eax, DWORD PTR pcw
    fldcw [eax]
    // compute E = 1417.0
    fldl // 1 in st(0)
    mov eax, DWORD PTR pa10
    fdiv DWORD PTR [eax] // 1/10 in st(0)
    fldl // 1 in st(0), 1/10 in st(1)
    mov eax, DWORD PTR pa3
    fdiv DWORD PTR [eax] // 1/3 in st(0), 1/10 in st(1)
    fdivp st(1), st(0) // 3/10 in st(0)
    fldl // 1 in st(0), 3/10 in st(1)
    fxch // 3/10 in st(0), 1 in st(1)
    fdivp st(1), st(0) // 10/3 in st(0)
    mov eax, DWORD PTR pa3
    fld DWORD PTR [eax] // 3 in st(0), 10/3 in st(1)
    mov eax, DWORD PTR pa10
    fdiv DWORD PTR [eax] // 3/10 in st(0), 10/3 in st(1)
    faddp st(1), st(0) // 109/30 in st(0)
    mov eax, DWORD PTR pa11
    fdiv DWORD PTR [eax] // 109/330 in st(0)
    fldl // 1 in st(0), 109/330 in st(1)
    mov eax, DWORD PTR pa99
    fdiv DWORD PTR [eax] // 1/99 in st(0), 109/330 in st(1)
    fldl // 1 in st(0), 1/99 in st(1), 109/330 in st(2)
    fxch // 1/99 in st(0), 1 in st(1), 109/330 in st(2)
    fdivp st(1), st(0) // 99 in st(0), 109/330 in st(1)
    mov eax, DWORD PTR pa11
    fadd DWORD PTR [eax] // 110 in st(0), 109/330 in st(1)
    fmulp st(1), st(0) // 109/3 in st(0)
    mov eax, DWORD PTR pa39
    fmul DWORD PTR [eax] // 1417 in st(0)
    mov eax, DWORD PTR pres
    fst DWORD PTR [eax] // res from the FPU stack to memory, pop st(0)
    mov eax, DWORD PTR pt
    fstp TBYTE PTR [eax] // res from the FPU stack to memory, pop st(0)
}
printf ("res = %4.4x%4.4x%4.4x%4.4x%4.4x\n",
        t[4], t[3], t[2], t[1], t[0]); // t = 1417.0
printf ("res = %6.6f\n", res);
}

```

The output corresponds to a pure IEEE double-extended precision computation:

```
res = 4009b120000000000001
```

```
res = 1417.000000
```

This shows that the result is 1 ulp greater than 1417:

$$\text{res} = 1417 + 1 \text{ ulp} = 1417 + 2^{10-63} = 1417 - 2^{-53}$$

The relative error corresponding to the absolute error $e = -2^{-53}$ is

$$\varepsilon_3 = 1.0842 \cdot 10^{-19}$$

Thus, the relative errors for single precision, double precision, and double-extended precision are respectively:

$$\varepsilon_1 = 8.6147 \cdot 10^{-8} > \varepsilon_2 = 3.2092 \cdot 10^{-16} > \varepsilon_3 = 1.0842 \cdot 10^{-19}$$

The same ratio between the relative errors is likely to be maintained after thousands (or more) computation steps, but the magnitude of the relative errors becomes increasingly larger. The best computation model can be chosen if the algorithm is known in detail, either by experimentation, or by theoretical numerical analysis.

6 Conclusion

The FPU instructions offer the richest set of operations for complex floating-point computations, involving different precisions, trigonometric, logarithmic or exponential function calls, and mixed integer computations, BCD computations (useful in financial applications), and floating-point computations. Yet, whenever similar floating-point calculations need to be performed on a large number of data items, the SIMD model can provide significant performance improvements. When the range of the single precision format is sufficient in such cases, the SSE instructions are best to use. When the range of the double precision format is necessary, the SSE2 instructions are recommended.

The IA-32 FPU architecture, SSE extensions, and SSE2 extensions all comply with the mandatory requirements of the IEEE Standard for floating-point computations. For compliance with the recommendations of the standard, software extensions are necessary, which is acceptable (for example, software emulation of excepting SSE extensions or of SSE2 extensions is necessary before invoking a user floating-point software exception handler). Besides implementing all the IEEE Standard's recommendations, the IA-32 floating-point architecture adds a number of features, some of which have become de facto industry standards. For example, the flush-to-zero mode allows for performance increases at the cost of minor accuracy losses. Another example is allowing higher precision results to be created from lower precision operands. As a whole, the Intel architecture offers a very good combination of accuracy, performance, and flexibility for floating-point calculations.