



# Intel® C++ Floating-point Operations

---

Document Number: 315891-001US

## Disclaimer and Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL(R) 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, life sustaining, critical control or safety systems, or in nuclear facility applications. Intel may make changes to specifications and product descriptions at any time, without notice.

The software described in this document may contain software defects which may cause the product to deviate from published specifications. Current characterized software defects are available on request.

This document as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

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

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skool, Sound Mark, The Journey Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

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

Copyright (C) 1996-2007, Intel Corporation.

Portions Copyright (C) 2001, Hewlett-Packard Development Company, L.P.

## Table of Contents

Overview: Floating-point Operations .....	1
Floating-point Options Quick Reference .....	1
Understanding Floating-point Operations .....	4
Using the -fp-model or /fp Option .....	4
Floating-point Optimizations .....	7
Programming Objectives of Floating-point Applications .....	9
Denormal Numbers.....	10
Floating-point Environment.....	11
Setting the FTZ and DAZ Flags .....	12
Tuning Performance .....	13
Handling Floating-point Array Operations in a Loop Body .....	14
Reducing the Impact of Denormal Exceptions .....	14
Avoiding Mixed Data Type Arithmetic Expressions .....	15
Using Efficient Data Types .....	16
Index .....	17

## Overview: Floating-point Operations

This section introduces the floating-point support in the Intel® C++ Compiler and provides information about using floating-point operations in your applications.

The following table lists some possible starting points:

If you are trying to...	Then start with...
Understand the programming objectives of floating-point applications	Programming Objectives of Floating-Point Applications
Use the <code>-fp-model</code> (Linux* and Mac OS*) or <code>/fp</code> (Windows*) option	Using the <code>-fp-model</code> or <code>/fp</code> Option
Set the flush-to-zero (FTZ) or denormals-are-zero (DAZ) flags	Setting the FTZ and DAZ Flags
Tuning the performance of floating-point applications	Overview: Tuning Performance of Floating-Point Applications

## Floating-point Options Quick Reference

The Intel® Compiler provides various options for you to optimize floating-point calculations with varying degrees of accuracy and predictability on different Intel architectures. This topic lists these compiler options and provides information about their supported architectures and operating systems.

Linux* and Mac OS*	Windows*	Description
<b>IA-32, Intel® 64, and IA-64 architectures</b>		
<code>-fp-model</code>	<code>/fp</code>	Specifies semantics used in floating-point calculations. Values are <code>precise</code> , <code>fast</code> , <code>strict</code> , <code>source</code> , <code>double</code> , <code>extended</code> , and <code>except</code> . There are extended values for <code>safe</code> and <code>except</code> . <ul style="list-style-type: none"> <li><code>-fp-model</code> compiler option</li> </ul>
<code>-fp-speculation</code>	<code>/Qfp-speculation</code>	Specifies the speculation mode for floating-point operations. Values are <code>fast</code> , <code>safe</code> , <code>strict</code> , and <code>off</code> . <ul style="list-style-type: none"> <li><code>-fp-speculation</code> compiler option</li> </ul>
<code>-prec-div</code>	<code>/Qprec-div</code>	Attempts to use slower but more accurate implementation of floating-point divide. Use this option to disable the divide optimizations in cases where it is important to maintain the full range and precision for

Linux* and Mac OS*	Windows*	Description
		<p>floating-point division. Using this option results in greater accuracy with some loss of performance.</p> <p>Specifying <code>-no-prec-div</code> (Linux and Mac OS) or <code>/Qprec-div-</code> (Windows) enables optimizations that result in slightly less precise results than full IEEE division.</p> <ul style="list-style-type: none"> <li>• <code>-prec-div</code> compiler option</li> </ul>
No equivalent	<code>/Qlong-double</code>	Changes the default size of the long double data type.
<code>-complex-limited-range</code>	<code>/Qcomplex-limited-range</code>	<p>Enables the use of basic algebraic expansions of some arithmetic operations involving data of type <code>COMPLEX</code>. This can cause performance improvements in programs that use a lot of <code>COMPLEX</code> arithmetic. Values at the extremes of the exponent range might not compute correctly.</p> <ul style="list-style-type: none"> <li>• <code>-complex-limited-range</code> compiler option</li> </ul>
<code>-ftz</code>	<code>/Qftz</code>	<p>The default behavior depends on the architecture. Refer to the following topic for details:</p> <ul style="list-style-type: none"> <li>• <code>-ftz</code> compiler option</li> </ul>
<b>IA-32 and Intel® 64 Architectures Only</b>		
<code>-prec-sqrt</code>	<code>/Qprec-sqrt</code>	<p>Improves the accuracy of square root implementations, but using this option may impact speed.</p> <ul style="list-style-type: none"> <li>• <code>-prec-sqrt</code> compiler option</li> </ul>
<code>-pc</code>	<code>/Qpc</code>	<p>Changes the floating point significand precision. Use this option when compiling applications.</p> <p>The application must use <code>main()</code> as the entry point, and you must compile the source file containing <code>main()</code> with this option.</p> <ul style="list-style-type: none"> <li>• <code>-pc</code> compiler option</li> </ul>
<code>-rcd</code>	<code>/Qrcd</code>	Disables rounding mode changes for floating-point-to-

Linux* and Mac OS*	Windows*	Description
		integer conversions. <ul style="list-style-type: none"> <li>• -rcd compiler option</li> </ul>
-fp-port	/Qfp-port	Causes floating-point values to be rounded to the source precision at assignments and casts. <ul style="list-style-type: none"> <li>• -fp-port compiler option</li> </ul>
-mp1	/Qprec	This option rounds floating-point values to the precision specified in the source program prior to comparisons. It also implies <code>-prec-div</code> and <code>-prec-sqrt</code> (Linux and Mac OS) or <code>/Qprec-div</code> and <code>/Qprec-sqrt</code> (Windows). This option has less impact to performance and disables fewer optimizations than the <code>-fp-model precise</code> (Linux and Mac OS) or <code>/fp:precise</code> (Windows) option. <ul style="list-style-type: none"> <li>• -mp1 compiler option</li> </ul>
<b>IA-64 Architecture Only</b>		
-IPF-fma	/QIPF-fma	Enables or disables the contraction of floating-point multiply and add/subtract operations into a single operation. <ul style="list-style-type: none"> <li>• -IPF-fma compiler option</li> </ul>
-IPF-fp-speculation	/QIPF-fp-speculation	Deprecated. Instructs the compiler to speculate on floating-point operations. Use <code>-fp-speculate</code> (Linux and Mac OS) or <code>/Qfp-speculate</code> (Windows) instead. <ul style="list-style-type: none"> <li>• -IPF-fp-speculation compiler option</li> </ul>
-IPF-fp-relaxed	/QIPF-fp-relaxed	Enables use of faster but slightly less accurate code sequences for math functions, such as the <code>sqrt()</code> function and the divide operation. As compared to strict IEEE* precision, using this option slightly reduces the accuracy of floating-point calculations performed by these functions, usually limited to the least significant binary digit. <ul style="list-style-type: none"> <li>• -IPF-fp-relaxed compiler option</li> </ul>

## Understanding Floating-point Operations

This section provides an understanding of the floating-point operations that can be performed using the Intel® C++ Compiler for Windows.

### Using the `-fp-model` or `/fp` Option

The `-fp-model` (Linux\* and Mac OS\*) or `/fp` (Windows\*) option allows you to control the optimizations on floating-point data. You can use this option to tune the performance, level of accuracy, or result consistency across platforms for floating-point applications.

For applications that do not require support for denormalized numbers, the `-fp-model` or `/fp` option can be combined with the `-ftz` (Linux and Mac OS) or `/Qftz` (Windows) option to flush denormalized results to zero in order to obtain improved runtime performance on processors based on:

- IA-32 and Intel® 64 architectures
- IA-64 architecture

You can use keywords to specify the semantics to be used. Possible values of the keywords are as follows:

Keyword	Description
<code>precise</code>	Enables value-safe optimizations on floating-point data.
<code>fast [=1 2]</code>	Enables more aggressive optimizations on floating-point data.
<code>strict</code>	Enables <code>precise</code> and <code>except</code> , disables contractions, and enables <code>pragma stdc fenv_access</code> .
<code>source</code>	Rounds intermediate results to source-defined precision and enables value-safe optimizations.
<code>double</code>	Rounds intermediate results to 53-bit (double) precision and enables value-safe optimizations.
<code>extended</code>	Rounds intermediate results to 64-bit (extended) precision and enables value-safe optimizations.
<code>[no-]except</code> (Linux and Mac OS) or <code>except [-]</code> (Windows)	Determines whether floating-point exception semantics are used.

The default value of the option is `-fp-model fast=1` or `/fp:fast=1`, which means that the compiler uses more aggressive optimizations on floating-point calculations.

Several examples are provided to illustrate the usage of the keywords. These examples show:

- A small example of source code
- The semantics that are used to interpret floating-point calculations in the source code
- One or more possible ways the compiler may interpret the source code

#### Note

- The same source code is considered in all the included examples.
- There are several ways the compiler may interpret the code; we show just some of these possibilities.

### [-fp-model fast or /fp:fast](#)

Example source code:

```
float t0, t1, t2;
...
t0 = 4.0f + 0.1f + t1 + t2;
```

When this option is specified, the compiler applies the following semantics:

- Additions may be performed in any order
- Intermediate expressions may use single, double, or extended double precision
- The constant addition may be pre-computed, assuming the default rounding mode

Using these semantics, the following shows some possible ways the compiler may interpret the original code:

```
float t0, t1, t2;
...
t0 = (float)((double)t1 + (double)t2) + 4.1f;
```

```
float t0, t1, t2;
...
t0 = (t1 + t2) + 4.1f;
```

```
float t0, t1, t2;
...
t0 = (t1 + 4.1f) + t2;
```

### [-fp-model extended or /fp:extended](#)

This setting is equivalent to `-fp-model precise` on Linux systems based on the IA-32 architecture and `-fp-model precise` or `/fp:precise` on systems based on the IA-64 architecture.

Example source code:

```
float t0, t1, t2;
...
```

## Intel® C++ Floating-point Operations

```
t0 = 4.0f + 0.1f + t1 + t2;
```

When this option is specified, the compiler applies the following semantics:

- Additions are performed in program order
- Intermediate expressions use extended double precision
- The constant addition may be pre-computed, assuming the default rounding mode

Using these semantics, the following shows a possible way the compiler may interpret the original code:

```
float t0, t1, t2;  
...  
t0 = (float)((long double)4.1f + (long double)t1) + (long double)t2);
```

### [-fp-model source or /fp:source](#)

This setting is equivalent to `-fp-model precise` or `/fp:precise` on systems based on the Intel® 64 architecture.

Example source code:

```
float t0, t1, t2;  
...  
t0 = 4.0f + 0.1f + t1 + t2;
```

When this option is specified, the compiler applies the following semantics:

- Additions are performed in program order, taking into account any parentheses
- Intermediate expressions use the precision specified in the source code
- The constant addition may be pre-computed, assuming the default rounding mode

Using these semantics, the following shows a possible way the compiler may interpret the original code:

```
float t0, t1, t2;  
...  
t0 = ((4.1f + t1) + t2);
```

### [-fp-model double or /fp:double](#)

This setting is equivalent to `-fp-model precise` or `/fp:precise` on Windows systems based on the IA-32 architecture.

Example source code:

```
float t0, t1, t2;  
...  
t0 = 4.0f + 0.1f + t1 + t2;
```

When this option is specified, the compiler applies the following semantics:

- Additions are performed in program order
- Intermediate expressions use double precision
- The constant addition may be pre-computed, assuming the default rounding mode

Using these semantics, the following shows a possible way the compiler may interpret the original code:

```
float t0, t1, t2;
...
t0 = (float)((double)4.1f + (double)t1) + (double)t
```

### [-fp-model strict or /fp:strict](#)

Example source code:

```
float t0, t1, t2;
...
t0 = 4.0f + 0.1f + t1 + t2;
```

When this option is specified, the compiler applies the following semantics:

- Additions are performed in program order, taking into account any parentheses
- Expression evaluation matches expression evaluation under keyword `precise`.
- The constant addition will not be pre-computed, because there is no way to tell what rounding mode will be active when the program runs.

Using these semantics, the following shows a possible way the compiler may interpret the original code:

```
float t0, t1, t2;
...
t0 = (float)((long double)4.0f + (long double)0.1f) +
      (long double)t1) + (long double)t2);
```

### See Also

- `-fp-model` compiler option

## Floating-point Optimizations

Application performance is an important goal of the Intel® Compilers, even at default optimization levels. A number of optimizations involve transformations, such as evaluation of constant expressions at compiler time, hoisting invariant expressions out of loops, or changes in the order of evaluation of expressions. These optimizations usually help the compiler produce most efficient code possible. However, this may not be true for floating-point applications, because some optimizations may affect accuracy, reproducibility, and performance.

## Intel® C++ Floating-point Operations

Some optimizations are not consistent with strict interpretation of the ANSI or ISO standards for C and C++, which can result in differences in rounding and small variants in floating-point results that may be more or less accurate than the ANSI-conformant result.

Intel Compilers provide the `-fp-model` (Linux\* and Mac OS\*) or `/fp` (Windows\*) option, which allows you to control the optimizations performed when you build an application. The option allows you to specify the compiler rules for:

- Value safety: Whether the compiler may perform transformations that could affect the result. For example, in the SAFE mode, the compiler won't transform  $x/x$  to  $1.0$ . The UNSAFE mode is the default.
- Floating-point expression evaluation: How the compiler should handle the rounding of intermediate expressions. For example, when double precision is specified, the compiler may transform the statement `t0=4.0f+0/1f+t1+t2;` to `t0=(float)(4.1+(double)t1+(double)t2);`
- Floating-point contractions: Whether the compiler should generate floating-point multiply-add (FMA) on processors based on the IA-64 architecture. When enabled, the compiler may generate FMA for combined multiply/add; when disabled, the compiler must generate separate multiply/add with intermediate rounding.
- Floating-point environment access: Whether the compiler must account for the possibility that the program might access the floating-point environment, either by changing the default floating-point control settings or by reading the floating-point status flags. This is disabled by default. You can use the `-fp-model:strict` (Linux and Mac OS) `/fp:strict` (Windows) option to enable it.
- Precise floating-point exceptions: Whether the compiler should account for the possibility that floating-point operations might produce an exception. This is disabled by default. You can use `-fp-model:strict` (Linux and Mac OS) or `/fp:strict` (Windows); or `-fp-model:except` (Linux and Mac OS) or `/fp:except` (Windows) to enable it.

Consider the following example:

```
double a=1.5;
int x=0;
...
try {
    int t0=a; //raises inexact
    x=1;
    a*=2;
} except(1) {
    printf("SEH Exception: x=%d\n", x);
}
```

Without precise floating-point exceptions, the result is SEH Exception: x=1; with precision floating-point exceptions, the result is SEH Exception: x=0.

The following table describes the impact of different keywords of the option on compiler rules and optimizations:

Keyword	Value Safety	Floating-Point Expression Evaluation	Floating-Point Contractions	Floating-Point Environment Access	Precise Floating-Point Exceptions
<code>precise</code> <code>source</code> <code>double</code> <code>extended</code>	Safe	Varies Source Double Extended	Yes	No	No
<code>strict</code>	Safe	Varies	No	Yes	Yes
<code>fast=1</code> (default)	Unsafe	Unknown	Yes	No	No
<code>fast=2</code>	Very unsafe	Unknown	Yes	No	No
<code>except</code> <code>except-</code>	Unaffected Unaffected	Unaffected Unaffected	Unaffected Unaffected	Unaffected Unaffected	Yes No

#### Note

It is illegal to specify the `except` keyword in an unsafe safety mode.

Based on the objectives of an application, you can choose to use different sets of compiler options and keywords to enable or disable certain optimizations, so that you can get the desired result.

## Programming Objectives of Floating-point Applications

In general, the programming objectives of the floating-point applications fall into the following categories:

- **Accuracy:** The application produces that results that are close to the correct result.
- **Reproducibility and portability:** The application produces results that are consistent across different runs, different set of build options, different compilers, different platforms, and different architectures.
- **Performance:** The application produces the most efficient code possible.

Based on the goal of an application, you will need to balance the tradeoffs among these objectives. For example, if you are developing a 3D graphics engine, then performance can be the most important factor to consider, and reproducibility and accuracy can be your secondary concerns.

Intel® Compiler provides appropriate compiler options, such as the `-fp-model` (Linux\* and Mac OS\*) or `/fp` (Windows\*) option, which allows you to tune your

applications based on specific objectives. The compiler processes the code differently when you specify different compiler options. Take the following code as an example:

```
float t0, t1, t2;  
...  
t0=t1+t2+4.0f+0.1f;
```

If you specify the `-fp-model extended` (Linux and Mac OS) or `/fp:extended` (Windows) option in favor of accuracy, the compiler generates the following assembly code:

```
fld     DWORD PTR t1  
fadd   DWORD PTR t2  
fadd   DWORD PTR Cnst4.0  
fadd   DWORD PTR Cnst0.1  
fstp   DWORD PTR _t0
```

If you specify the `-fp-model source` (Linux and Mac OS) or `/fp:source` (Windows) option in favor of reproducibility and portability, the compiler generates the following assembly code:

```
movss  xmm0, DWORD PTR t1  
addss  xmm0, DWORD PTR t2  
addss  xmm0, DWORD PTR Cnst4.0  
addss  xmm0, DWORD PTR Cnst0.1  
movss  DWORD PTR _t0, xmm0
```

If you specify the `-fp-model fast` (Linux and Mac OS) or `/fp:fast` (Windows) option in favor of performance, the compiler generates the following assembly code:

```
movss  xmm0, DWORD PTR Cnst4.1  
addss  xmm0, DWORD PTR t1  
addss  xmm0, DWORD PTR t2  
movss  DWORD PTR _t0, xmm0
```

In the real world, an application may be much more complicated. You should select appropriate compiler options by carefully consider your programming objectives and balance the tradeoffs among these objectives.

## Denormal Numbers

A normalized number is a number for which both the exponent (including offset) and the most significant bit of the mantissa are non-zero. For such numbers, all the bits of the mantissa contribute to the precision of the representation.

The smallest normalized single precision floating-point number greater than zero is about  $1.1754943 \times 10^{-38}$ . Smaller numbers are possible, but those numbers must be represented with a zero exponent and a mantissa whose leading bit(s) are zero, which leads to a loss of precision. These numbers are called denormalized numbers; denormals (newer specifications refer to these as subnormal numbers).

Denormal computations use both hardware or operating system resources to handle them, which can cost hundreds of clock cycles.

- Denormal computations take much longer to calculate on IA-32 and Intel® 64 processors than normal computations.
- Denormals are computed in software on processors based on the IA-64 architecture, and the computation usually requires hundreds of clock cycles, which results in excessive kernel time.

There are several ways to handle denormals and increase the performance of your application:

- Scale the values into the normalized range.
- Use a higher precision data type with a larger dynamic range.
- Flush denormals to zero.

### See Also

- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*
- *Intel® Itanium® Architecture Software Developer's Manual, Volume 1: Application Architecture*
- Institute of Electrical and Electronics Engineers, Inc\*. (IEEE) web site for information about the current floating-point standards and recommendations.

## Floating-point Environment

The floating-point environment is a collection of registers that control the behavior of the floating-point machine instructions and indicate the current floating-point status. The floating-point environment can include rounding mode controls, exception masks, flush-to-zero (FTZ) controls, exception status flags, and other floating-point related features.

For example, on IA-32 and Intel® 64 architectures, bit 15 of the MXCSR register enables the flush-to-zero mode, which controls the masked response to an single-instruction multiple-data (SIMD) floating-point underflow condition.

The floating-point environment affects most floating-point operations; therefore, correct configuration to meet your specific needs is important. For example, the exception mask bits define which exceptional conditions will be raised as exceptions by the processor. In general, the default floating-point environment is set by the operating system. You don't need to configure the floating-point environment unless the default floating-point environment does not suit your needs.

There are several methods available if you want to modify the default floating-point environment. For example, you can use inline assembly, compiler built-in functions, and library functions.

Changing the default floating-point environment affects runtime results only. This does not affect any calculations which are pre-computed at compile time.

If strict reproducibility and consistency are important do not change the floating point environment without also using either `fp-model strict` (Linux or Mac OS\*) or `/fp:strict` (Windows\*) option or `pragma fenv_access`.

**See Also**

- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*
- *Intel® Itanium® Architecture Software Developer's Manual, Volume 1: Application Architecture*

**Setting the FTZ and DAZ Flags**

In Intel® processors, the flush-to-zero (FTZ) and denormals-are-zero (DAZ) flags in the MXCSR register are used to control floating-point calculations. When the FTZ and DAZ flags are enabled, the Single Instructions and Multiple Data (SIMD) floating-point computation can be accelerated, thus improving the performance of the application.

You can use the `-ftz` (Linux\* and Mac OS\*) or `/Qftz` (Windows\*) option to flush denormal results to zero when the application is in the gradual underflow mode. This option may improve performance if the denormal values are not critical to your application's behavior.

The `-ftz` or `/Qftz` option sets or resets the FTZ and the DAZ hardware flags. The following table describes how the compiler process denormal values based on the status of the FTZ and DAZ flags:

Flag	When set to ON, the compiler...	When set to OFF, the compiler...	Supported Architectures
FTZ	Sets denormal results from floating-point calculations to zero.	Does not change the denormal results.	IA-64 Intel® 64
DAZ	Treats denormal values used as input to floating-point instructions as zero.	Does not change the denormal instruction inputs.	Intel® 64

- FTZ and DAZ are not supported on all IA-32 architectures. On systems based on the IA-64 architecture, FTZ always works, while on systems based on the IA-32 and Intel® 64 architectures, it only applies to SSE instructions. Hence if your application happened to generate denormals using x87 instructions, FTZ does not apply.
- DAZ and FTZ flags are not compatible with IEEE Standard 754, so you should only consider enabling them when strict compliance to the IEEE standard is not required and application performance has higher priority than application accuracy.

Options `-ftz` and `/Qftz` are performance options. Setting these options does not guarantee that all denormals in a program are flushed to zero. They only cause denormals generated at run time to be flushed to zero.

When `-ftz` or `/Qftz` is used in combination with an SSE-enabling option on systems based on the IA-32 architecture (for example, `xW` or `QxW`), the compiler will insert

code in the main routine to set FTZ and DAZ. When `-ftz` or `/Qftz` is used without such an option, the compiler will insert code to conditionally set FTZ/DAZ based on a run-time processor check. `-no-ftz` (Linux and Mac OS) or `/Qftz-` (Windows) will prevent the compiler from inserting any code that might set FTZ or DAZ.

The `-ftz` or `/Qftz` option only has an effect when the main program is being compiled. It sets the FTZ/DAZ mode for the process. The initial thread and any threads subsequently created by that process will operate in the FTZ/DAZ mode.

On systems based on the IA-64 architecture, optimization option O3 sets `-ftz` and `/Qftz`; optimization option O2 sets `-no-ftz` (Linux) and `/Qftz-` (Windows). On systems based on the IA-32 and Intel® 64 architectures, every optimization option O level, except O0, sets `-ftz` and `/Qftz`.

If this option produces undesirable results of the numerical behavior of your program, you can turn the FTZ/DAZ mode off by using `-no-ftz` or `/Qftz-` in the command line while still benefiting from the O3 optimizations.

For some non-Intel processors, you can set the flags manually with the following macros:

Feature	Examples
Enable FTZ	<code>_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON)</code>
Enable DAZ	<code>_MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON)</code>

The prototypes for these macros are in `xmmintrin.h` (FTZ) and `pmmintrin.h` (DAZ).

### See Also

- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*

## Tuning Performance

This section describes several programming guidelines that can help you improve the performance of a floating-point applications:

- Avoid exceeding representable ranges during computation; handling these cases can have a performance impact.
- Use a single precision type (for example, `float`) unless the extra precision obtained through `double` or `long double` is required. Greater precision types increase memory size and bandwidth requirements.
- Reduce the impact of denormal exceptions for all supported architectures.
- Avoid mixed data type arithmetic expressions.
- Use efficient data types.

## Handling Floating-point Array Operations in a Loop Body

The statements within the loop body may contain float operations (typically on arrays). The following arithmetic operations are supported: addition, subtraction, multiplication, division, negation, square root, MAX, MIN, and mathematical functions such as SIN and COS.

Operation on `DOUBLE PRECISION` types is not valid, unless optimizing for a Pentium® 4 and Intel® Xeon® processors system, using the `-xW` (Linux\* and Mac OS\*) or `/QxW` (Windows\*) or `-axW` (Linux) or `/QaxW` (Windows) compiler option.

Note that the special `__m64` and `__m128` datatypes are not vectorizable. The loop body cannot contain any function calls. Use of the Streaming SIMD Extensions intrinsics (`_mm_add_ps`) are not allowed.

## Reducing the Impact of Denormal Exceptions

Denormalized floating-point values are those that are too small to be represented in the normal manner; for example, the mantissa cannot be left-justified. Denormal values require hardware or operating system interventions to handle the computation, so floating-point computations that result in denormal values may have an adverse impact on performance.

There are several ways to handle denormals to increase the performance of your application:

- Scale the values into the normalized range
- Use a higher precision data type with a larger dynamic range
- Flush denormals to zero

For example, you can translate them to normalized numbers by multiplying them using a large scalar number, doing the remaining computations in the normal space, then scaling back down to the denormal range. Consider using this method when the small denormal values benefit the program design.

Consider using a higher precision data type with a larger dynamic range. For example, converting variables declared as `float` to be declared as `double`.

Understand that making the change has the potential to cause your program to slow down, storage requirements will increase, which increases the amount of time loading and storing data from memory; it can also decrease the potential throughput of SSE operations.

If you change the declaration of a variable you might also need to change the libraries you call to use the variable; for example, `cosd()` instead of `cos()`. Another strategy that might result in increased performance is to increase the amount of precision of intermediate values using the `-fp-model [double|extended]` option; however, if you increased precision as the solution to slow performance caused by denormal numbers you must verify the resulting changes actually increase performance.

Finally, In many cases denormal numbers be treated safely as zero without adverse effects on program results. Depending on the target architecture, use flush-to-zero (FTZ) options.

## IA-32 and Intel® 64 Architectures

These architectures take advantage of the FTZ and DAZ (denormals-are-zero) capabilities of Streaming SIMD Extensions (SSE), Streaming SIMD Extensions 2 (SSE2), and Streaming SIMD Extensions 3 (SSE3), and Supplemental Streaming SIMD Extensions 3 (SSSE3) instructions.

By default, the compiler for the IA-32 architecture generates code that will run on machines that do not support SSE instructions. The compiler implements floating-point calculations using the x87 floating-point unit, which does not benefit from the FTZ and DAZ settings. You can use the `-x` (Linux\* and Mac OS\*) or `/Qx` (Windows\*) option to enable the compiler to implement floating-point calculations using the SSE and SSE2 instructions. The compiler for the Intel® 64 architecture generates SSE2 instructions by default.

The FTZ and DAZ modes are enabled by default when you compile the source file containing `main()` using the Intel Compiler. The compiler generates a call to a library routine that performs a runtime processor check. The FTZ and DAZ modes are set provided that the modes are available for the machine on which the program is running.

## IA-64 Architecture

Enable the FTZ mode by using the `-ftz` (Linux and Mac OS) or `/Qftz` (Windows) option on the source file containing `main()`. The `-O3` (Linux and Mac OS) or `/O3` (Windows) option automatically enables `-ftz` or `/Qftz`.

### Note

After using flush-to-zero, ensure that your program still gives correct results when treating denormalized values as zero.

### See Also

- Setting the FTZ and DAZ Flags
- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*

## Avoiding Mixed Data Type Arithmetic Expressions

Avoid mixing integer and floating-point (`float`, `double`, or `long double`) data in the same computation. Expressing all numbers in a floating-point arithmetic expression (assignment statement) as floating-point values eliminates the need to convert data between fixed and floating-point formats. Expressing all numbers in an integer arithmetic expression as integer values also achieves this. This improves runtime performance.

For example, assuming that `I` and `J` are both `int` variables, expressing a constant number (2.) as an integer value (2) eliminates the need to convert the data. The following examples demonstrate inefficient and efficient code.

**Example: Inefficient Code**

```
int I, J;  
I = J / 2.;
```

**Example: Efficient Code**

```
int I, J;  
I = J / 2;
```

You can use different sizes of the same general data type in an expression with minimal or no effect on run-time performance. For example, using `float`, `double`, and `long double` floating-point numbers in the same floating-point arithmetic expression has minimal or no effect on run-time performance. However, this practice of mixing different sizes of the same general data type in an expression can lead to unexpected results due to operations being performed in a lower precision than desired.

## Using Efficient Data Types

In cases where more than one data type can be used for a variable, consider selecting the data types based on the following hierarchy, listed from most to least efficient:

- `char`
- `short`
- `int`
- `long`
- `long long`
- `float`
- `double`
- `long double`

However, keep in mind that in an arithmetic expression, you should avoid mixing integer and floating-point data.

You can use integer data types (`int`, `int long`, etc.) in loops to improve floating point performance. Convert the data type to integer data types, process the data, then convert the data to the old type.

## Index

/

/fp compiler option.....1, 4

/Qftz compiler option.....12

### A

architectures

    coding guidelines for.....13

array.....14

array operation.....14

avoid

    inefficient data types .....15

    mixed arithmetic expressions .....15

### D

data types

    efficiency .....16

DAZ flag .....12

denormal exceptions .....14

denormal numbers.....10

denormals.....10

### E

efficiency .....13

efficient data types.....16

### F

floating-point.....1

floating-point array operation .....14

floating-point exceptions

    denormal exceptions.....14

-fp-model compiler option .....1, 4

-ftz compiler option .....12

FTZ flag.....12

### M

maintainability.....13

MXCSR register.....12

### O

overview.....1

### P

performance.....13

### R

run-time performance

    improving.....13

### S

subnormal numbers.....10