Technology Guide

intel®

Intel® AVX-512 - FP16 Instruction Set for Intel® Xeon® Processor-Based Products

Author and Editor

Daniel Towner

Contributors

Steven Wood

Phoebe Wang

Marius Cornea

Cristina Anderson

Amit Gradstein

1 Introduction

This document describes the new FP16 instruction set architecture (ISA) for Intel® Advanced Vector Extensions 512 (Intel® AVX-512) that is added to 4th generation Intel® Xeon® Scalable processors. The new ISA supports a wide range of general-purpose numeric operations for 16-bit half-precision IEEE-754 floating-point and complements the existing 32-bit and 64-bit floating-point instructions already available in the Intel Xeon processor-based products. The new ISA also provides complex-valued native hardware support.

The new ISA is ideal for numeric operations where reduced precision can be used, such as signal and media processing. For example, wireless signal processing operations such as beamforming, precoding, and minimum mean squared error (MMSE) perform well with this ISA. This instruction set also works well with traditional signal processing, for example, with real or complex-valued Fast Fourier Transforms (FFTs). The advantage of using reduced precision in these cases is that because fewer bits are processed for each element, the overall compute throughput can be increased, allowing precision and speed to be traded against each other.

This document is part of the Network & Edge Platform Experience Kits.

Table of Contents

1 Introduction ... 1
1.1 Terminology ... 4
1.2 Reference Documentation ... 4

2 Overview ... 4

3 FP16 Numeric Instructions ... 5
3.1 Data Type Support ... 5
3.2 Overview of Intrinsics ... 6
3.3 Fundamental Complex-Valued Support ... 7
3.4 Using AVX-512 Bit Masks for Real-Valued Operations ... 7
3.5 Using AVX-512 Bit Masks for Complex-Valued Operations ... 8

4 Numerics ... 11
4.1 Introduction to FP16 Number Format ... 11
4.2 Observations on Representing Numbers in FP16 Format ... 11
4.3 Numeric Accuracy Guarantees ... 13
4.4 Handling Denormal Values ... 13
4.5 Embedded Rounding ... 13
4.6 Legacy FP16 Data Type Conversion ... 14
4.7 FP16 Conversions to and from Other Data Types ... 14
4.8 Approximation Instructions and Their Uses ... 15
4.8.1 Approximate Reciprocal ... 15
4.8.2 Approximate Division ... 15
4.8.3 Approximate Reciprocal Square Root ... 16
4.9 Approximate Square Root ... 16

5 Using Existing Intel AVX-512 Instructions to Augment FP16 Support ... 17
5.1 Using Existing Instructions to Extend Intel AVX-512 FP16 Intrinsics ... 17
5.2 Common Convenience Intrinsics ... 17

6 Using FP16 Without Hardware Support ... 18
6.1 Emulation of Targets with the FP16 Instruction Set ... 18
6.2 Compiler Synthesis of FP16 Instructions ... 18
6.2.1 Intermediate (or Excess) Precision ... 18
6.2.2 Controlling the Intermediate Precision ... 19
6.2.3 Fast Math Optimizations ... 20
6.2.4 Summary and Recommendations for FP16 Emulation or Synthesis ... 20

7 Math Library Support ... 20

8 Summary ... 21

Figures

Figure 1. Layout of a 128-Bit Register Representing Four Complex FP16 (CFP16) Values ... 6
Figure 2. Illustration of a Zero-Masked FP16 Add on Two 128-Bit Vectors ... 8
Figure 3. Illustration of a Masked Complex Multiplication ... 8
Figure 4. Illustration of Using a Real-Valued FP16 Vector Operation for Implementing a Masked Complex Addition ... 9
Figure 5. Illustration of Using a Real-Valued FP16 Vector Operation for Implementing a Masked Complex Addition ... 9
Figure 6. Comparison Operation Between Two Complex-Valued Vectors. The Mask Bits Are Generated Using a Real-Valued Comparison, and Then Adjacent Bits Combined Using AND ... 10
Figure 7. Function for Converting from a Real-Valued Mask to a Complex-Valued Mask By AND-Combining Adjacent Bits ... 10
Figure 8. Function for Converting from a Real-Valued Mask To a Complex-Valued Mask By OR-Combining Adjacent Bits ... 11
Figure 9. Bit Layout of Three Types of Floating-Point Formats ... 11
Figure 10. Landmark Numbers on the Real-Valued FP16 Axis ... 12
Figure 11. Heat Map Showing Relative ULP Error for Different Combinations of Divisor and Dividend Value Ranges ... 16
Figure 12. Function to Implement the 16-Bit Compress Operation on FP16 Vector Elements ... 17

Tables

Table 1. Terminology 4
Table 2. Reference Documents 4
Table 3. Supported FP16 Data Types 6
Table 4. Example Intrinsic Names 6
Table 5. Conjugation Instructions 7
Table 6. Useful or Interesting FP16 Numbers 12
Table 7. Rounding Modes 14
Table 8. Convenience Intrinsics 17

Document Revision History

Revision	Date	Description
001	October 2021	Initial release.
002	January 2022	Clarified naming of rounding modes and provided a faster code sequence for approximate square root.
003	May 2022	Clarified when embedded rounding is permitted. Revised the document for public release to Intel® Network Builders.
004	March 2024	Added description of using the FP16 data type without hardware support.

1.1 Terminology

Table 1. Terminology

Abbreviation	Description
CFP16	Complex-valued floating-point format comprising two FP16 values representing the real and imaginary values respectively. When used in SIMD, the individual real/imaginary values from each complex value are interleaved in the register.
Denormal	A subset of denormalized numbers that fill the underflow gap around zero in floating-point arithmetic.
FP16	Half precision 16-bit floating-point data format.
FP32	Single precision 32-bit floating-point data format
FP64	Double precision 64-bit floating-point data format
FFT	Fast Fourier Transform
IEEE 754-2019	The current IEEE Standard for Floating-Point Arithmetic used in Intel AVX-512 FP16
Intel® AVX/Intel® AVX2/Intel® AVX-512	Intel® Advanced Vector Extensions, Intel® Advanced Vector Extensions 2, Intel® Advanced Vector Extensions 512 Advanced Vector Extensions. Available in three major revisions, ranging from basic 256-bit SIMD support in AVX, through additional instructions and data types in Intel AVX2, and on to the most recent 512-bit support in Intel AVX-512.
Intel® AVX-512 FP16	A new ISA for handling half precision floating-point, added as an extension to Intel AVX-512.
Intrinsic	A function that can be called from a high-level language, such as C/C++, that gives direct access to the underlying ISA. Intrinsics allow the programmer to bypass the compiler and directly specify that a particular instruction be used.
ISA	Instruction Set Architecture
MMSE	Minimum Mean Squared Error
NaN	Not A Number. A way to represent a value that is undefined or unrepresentable. For example, the square root of a negative number would generate a NaN value.
Normal	A floating-point number that can be represented without leading zeros in its significand.
SIMD	Single instruction, multiple data. A way of packing several data elements into a single container and operating on them all at once.
SINR	Signal-to-Interference-plus-Noise Ratio
SSE	SIMD Streaming Extensions. The predecessor to AVX.

1.2 Reference Documentation

Table 2. Reference Documents

Reference	Source
Intel® AVX-512-FP16 Architecture Specification	https://software.intel.com/content/www/us/en/develop/download/intel-avx512-fp16-architecture-specification.html
Intel® 64 and IA-32 Architectures Optimization Reference Manual: Volume 1	https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf
Intel® 64 and IA-32 Architectures Software Developer Manuals	https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html
Intel® Intrinsics Guide, an online reference guide	https://software.intel.com/sites/landingpage/IntrinsicsGuide/
Intel® Software Development Emulator (Intel® SDE) Product Overview	https://www.intel.com/content/www/us/en/developer/articles/tool/software-development-emulator.html

2 Overview

In this document, we describe the addition of a new FP16 ISA for Intel AVX-512 into the Intel Xeon processor family to handle IEEE 754-2019[1] compliant half-precision floating-point operations (also known officially as binary16, or unofficially as FP16). This new instruction set is general-purpose and can be used for all numeric operations that could be reasonably expected, including numeric operations (add, subtract, divide, multiply), fused operations (for example, fused multiply-add), comparisons, conversions to and from other data types, and many more. Broadly, the new instruction set mirrors the floating-point support that is already available in Intel Xeon processors for 32-bit (FP32) and 64-bit (FP64), although there are a few exceptions to this,

[1] https://standards.ieee.org/standard/754-2019.html

which will be noted where appropriate. There is one notable new feature of FP16 when compared to existing FP32 and FP64 instruction sets: The addition of native complex-value support for interleaved FP16 data, which is useful in scientific computing and signal processing.

The two major advantages of using the new FP16 instruction set compared to other floating-point formats are increased execution throughput and reduced storage requirements. Half-precision floating-point values only require 16 bits for storing each value, as opposed to the 32 or 64 bits needed for other common IEEE floating-point formats. This allows FP16 to handle twice as many operations per clock cycle compared to FP32, and four times as many compared to FP64. Similarly, the reduced size means that more values can be stored in a given memory region compared to the other formats, increasing the effectiveness of the registers and the cache hierarchy. The disadvantages are the reduced range and precision. It is the responsibility of the programmer to decide whether this floating-point format is suitable for a certain application.

Half-precision floating-point is useful for building systems where the dynamic range of floating-point is required but a lower numeric precision can be easily tolerated and traded for higher compute performance. Typical applications for half-precision floating-point include signal processing, media or video processing, artificial intelligence, and machine learning.

Historically, limited support for half-precision data types was available in processors from the 3rd generation Intel® Core™ processor onwards, but the operations were restricted to converting between half-precision and FP32 values. On older platforms, all numeric operations had to be implemented using higher precision formats and down-converted on completion. Those instructions were useful for compatibility with other platforms (for example, Intel® GPUs), but did not realize the benefits in higher compute performance brought about in FP16.

IEEE FP16 is not the only 16-bit floating-point format. Another common type is bfloat16, which is primarily used in artificial intelligence and machine learning. Intel Xeon processors support some bfloat16 operations, including type conversions and a few limited numeric operations, but not the full range of general-purpose operations that are supported in FP16 for Intel AVX-512. This document describes only the instruction set relating to IEEE 754-2019.

This document covers both the general-purpose instruction set as well as the new complex-valued instructions. We then look at the numeric implications of using FP16 and discuss how to write optimal code sequences for some common operations.

The examples provided in this document use the intrinsic and data type support provided as part of the Intel® oneAPI DPC++ Compiler.

3 FP16 Numeric Instructions

FP16 is an instruction set extension that mirrors the existing support for other floating-point operations in Intel AVX-512 and makes it available in IEEE-754 FP16 (binary16) number format. It is a general-purpose instruction set, and features instructions that support all common operations that are required in typical numeric software applications. Briefly, the following classes of instructions are supported:

Fundamental IEEE numeric	Addition, subtraction, multiplication, division, square root
Fused	Fused (multiply-accumulate) operations covering fmadd, fmsub, negated fma, fmaddsub, and fmsubadd
Comparison	Minimum, maximum, compare-to-mask (for example, neq, lt, gt)
Conversions	Conversions to and from other common data types, including 16/32/64-bit integer and FP32/FP64 floating-point
Approximation	Fast, but approximate operations to support reciprocal and reciprocal-square-root.
Specialized	Significand (mantissa)/exponent manipulation, scaling, and rounded scaling
Complex	Native complex-value multiply and fused-multiply operations

We will now look at how to use these new instructions, the impact on performance, and the consequences of the reduced floating-point decision.

3.1 Data Type Support

Table 3 shows the new data types supported with the FP16 instruction set. In each case, the name of the equivalent type in C or C++ is provided.

Table 3. Supported FP16 Data Types

Type Format	C/C++ Type Name	Notes
Scalar	`_Float16`	Single 16-bit value stored in IEEE FP16 format
128-bit AVX register	`__m128h`	8 x FP16 values, or 4 x complex FP16 values (CFP16)
256-bit AVX2 register	`__m256h`	16 x FP16 values, or 8 x complex FP16 values (CFP16)
512-bit AVX-512 register	`__m512h`	32 x FP16 values, or 16 x complex FP16 values (CFP16)

The complex instructions operate on standard SIMD vector types, such as `__m128h`, but internally those instructions treat the register as sets of complex-valued pairs, as shown in Figure 1. Note that we shall refer to a complex pair of FP16 values as `CFP16'. The CFP16 type is laid out as though it were an array of two FP16 values, or a C++ type such as `std::complex<_Float16>`.

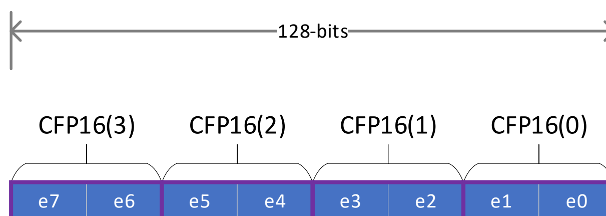

Figure 1. Layout of a 128-Bit Register Representing Four Complex FP16 (CFP16) Values

In the latest Intel OneAPI compilers, 16-bit floating-point literals can be created by suffixing a value with `f16`. For example:

```
_Float16 value = 12.34f16;
```

3.2 Overview of Intrinsics

In common with the intrinsics for other vector instruction sets, FP16 intrinsics take the following form:

```
result = _mmBITLENGTH_OPNAME_ELEMENTTYPE(arguments)
```

The bit-length can be 512, 256, or 128 bits, and in the 128-bits case the `BITLENGTH` field is empty. The `OPNAME` is a short descriptor or abbreviation of what the operation does (for example, add, sub, fmadd). The element type is `sh` for FP16 scalar (scalar-half), `ph` for a vector of FP16 values (packed-half), or `pch` for a vector of CFP16 values (packed-complex-half). Table 4 gives a few examples to illustrate the naming conventions.

Table 4. Example Intrinsic Names

Intrinsic Name	Description
`_mm_sub_sh`	Subtract a single scalar FP16 element from another scalar FP16 element
`_mm_add_ph`	Add a pair of 8xFP16 vector registers to form a result containing 8xFP16 outputs
`_mm256_add_ph`	Add a pair of 16xFP16 vector registers to form a result containing 16xFP16 outputs
`_mm512_add_ph`	Add a pair of 32xFP16 vector registers to form a result containing 32xFP16 outputs
`_mm256_fmadd_ph`	Multiply a pair of 16xFP16 vector registers and add the result to a third vector register of 16xFP16 values, forming a result containing 16xFP16 vector elements
`_mm512_rcp_ph`	Compute the reciprocal of a vector register containing 32xFP16 values, generating an output of another vector register containing 32xFP16 values.
`_mm256_fmadd_pch`	Compute the complex multiplication of 8xCFP16 (complex-FP16) values, adding the result to another such register, and generating a result containing 8xCFP16 elements.
`_mm512_conj_pch`	Compute the conjugate of a 512-bit register containing 16xCFP16 (complex-FP16) elements.

Note that `pch` complex operation intrinsics are only provided for multiply and fused-multiply-add operations since these require special hardware support. No intrinsics are provided for operations like addition, since the existing `add_ph` intrinsic behaves correctly for those without extra support requirements.

For a complete list of all the intrinsics provided as part of FP16, refer to the Intel® Intrinsics Guide.

In the remainder of this document where we give the names of intrinsics, we typically only show the 128-bit variant. Because AVX512-FP16 supports VL encoding, all three length variants of the intrinsics are available (that is, 128-bit, 256-bit, 512-bit).

3.3 Fundamental Complex-Valued Support

For complex-valued operations, the primary place where hardware support is provided is in multiplication. Complex multiplication requires several steps, and the new FP16 ISA accelerates those steps. Simpler operations, such as addition and subtraction, do not require explicit complex support since these can be handled using the other FP16 instructions (for example, addition of two complex numbers is just the addition of respective real and imaginary values from the two inputs, so `_mm_add_ph` can be directly used). Note that complex division is not supported in hardware as this is an uncommon operation, and it can be constructed from the hardware multiplier and complex multiplier support if required.

To illustrate the mechanics of how complex multiplication is supported, consider the following complex multiply:

```
(a + bi) * (c + di)
```

This operation can be refactored as follows:

```
(ac − bd) + (ad + bc)i
```

Note that to compute each of the real and imaginary components of the multiply a stand-alone multiply is used first, followed by a suitable fmadd/fmsub instruction. The hardware support for complex-valued multiplies uses these partial mul/fmadd instructions in sequence to perform the entire complex multiply. The hardware can schedule and route the data inside the processor to do this more quickly and efficiently than using an explicit instruction sequence to move the real and imaginary data into the correct places. Note however that each intermediate step produces a temporary FP16 answer, so the final result will have had an FP16 quantization step.

Using the symbols from the example above, a complex-fma (that is, accumulating against another complex number) can be implemented using the following refactoring:

```
((accReal + ac) − bd) + ((accImag + ad) + bc)i
```

This sequence is equivalent to two FMA operations being performed for each of the real/imag components.

The conjugate of a complex number is formed by negating its imaginary component. A common operation with conjugation is to multiply a complex number with a conjugate of another complex number. Conjugation in FP16 is supported using three classes of intrinsic, as illustrated in Table 5.

Table 5. Conjugation Instructions

Intrinsic Name	Description
`_mm_conj_pch`	Compute the conjugate of a register containing CFP16 (complex-FP16) elements by negating each imaginary value.
`_mm_fcmul_pch`	Compute the multiplication of a conjugated value with another complex value.
`_mm_fcmadd_pch`	Compute the multiplication of a conjugated value with another complex value, adding the result to a third complex-vector register.

Both the multiply and the FMA are able to perform the conjugation as part of the instruction operation itself. It is not necessary to conjugate the value first. For example, an `_mm_fcmul_pch` intrinsic is functionally equivalent to:

```
_mm_fcmul_pch(_mm_conj_pch(lhs), rhs)
```

But `_mm_fcmul_pc` will operate in fewer cycles than calling that sequence explicitly. When the compiler notices a separate conjugate and multiply intrinsics being used it fuses them into a single conjugate-multiply operation.

3.4 Using AVX-512 Bit Masks for Real-Valued Operations

FP16 is able to use the bit-mask features of Intel AVX-512 both to control when operations in a vector register take place, and to generate masks that store the results of performing tests on vector registers.

Masks allow execution of an instruction to be conditionally applied to selected elements of a vector register. Most instructions permit such a mask register to be supplied as part of the operation, where each bit within the mask corresponds to a different element of the vector register. If a given mask bit is set, then the instruction operates on the corresponding element. While if the bit is cleared, the operation is not performed and that element's output is replaced by another value. The cleared output value can either be taken from another source register, or it can be zeroed. The operation of a masked instruction is illustrated in Figure 2. Note that the operation only takes place where the 8-element mask has a corresponding bit set and all other outputs are zeroed.

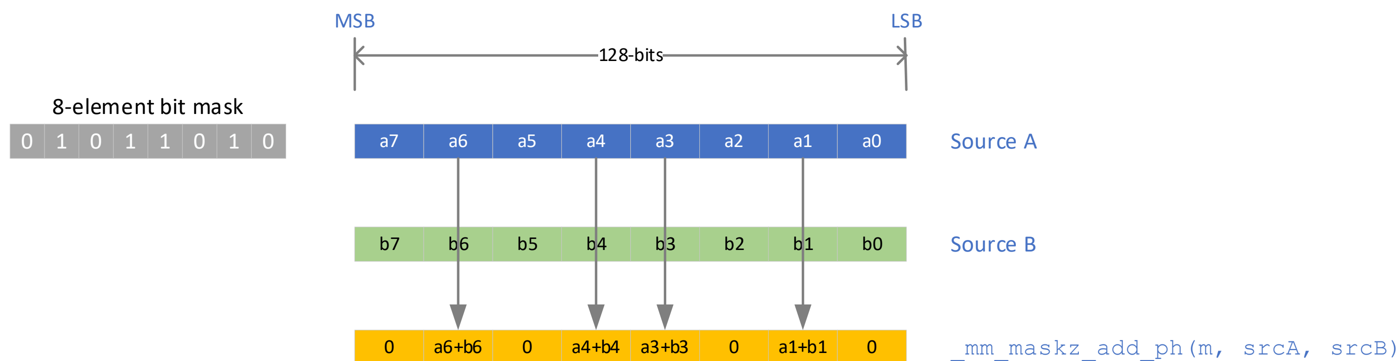

Figure 2. Illustration of a Zero-Masked FP16 Add on Two 128-Bit Vectors

Note that masks also control whether faults within an instruction are suppressed. If an operation generates a fault in a particular element, but the element's operation has been disabled by a zero bit in the mask, then the fault is not reported.

Some instructions in Intel AVX-512 can generate mask registers, and with FP16 these are normally the result of a comparison operation. For example, consider the following code snippet:

```
whichElementsAreLess = _mm512_cmp_ph_mask(lhs, rhs, _CMP_LT_OS);
```

In this example, every element of the left-hand vector is compared to see if it is less than the corresponding element in the right-hand vector. If the left element is less than the right element, then a 1 is generated in the mask bit output, otherwise a zero is emitted. This comparison instruction allows all the major binary comparison operations to be performed between two vectors.

The FP16 instruction also provides support to test for special values using the `_mm_fpclass_ph_mask` instruction. This instruction takes a special immediate value that directs the instruction to the numeric classes to look for in the vector register (for example, infinities, NaN, zero, denormal). This instruction is often used in combination with other instructions to remove special case values from a register and replace them with something different. For example, the following code snippet removes NaN values and replaces them with zero.

```
__mmask8 whichAreNan = _mm_fpclass_ph_mask(values, QUIET_NAN | SIGNAL_NAN);

__m128h valuesWithNoNan = _mm_mask_blend_ph(whichAreNan, values, __m128h());
```

In Intel AVX-512, there is a special instruction that does direct replacement of special values with known constants called `_mm_fixupimm_ps/pd`, but this is not available in FP16.

3.5 Using AVX-512 Bit Masks for Complex-Valued Operations

When a mask operation is applied to an intrinsic that operates directly on complex instruction data (for example, `_mm512_mask_fmadd_pch`), then each mask bit refers to a complex pair of FP16 values, not to the individual FP16 values. This is illustrated in Figure 3. Note that there are 8 FP16 elements, grouped into 4 CFP16 complex values. The 4 mask bits correspond to the 4 CFP16 values.

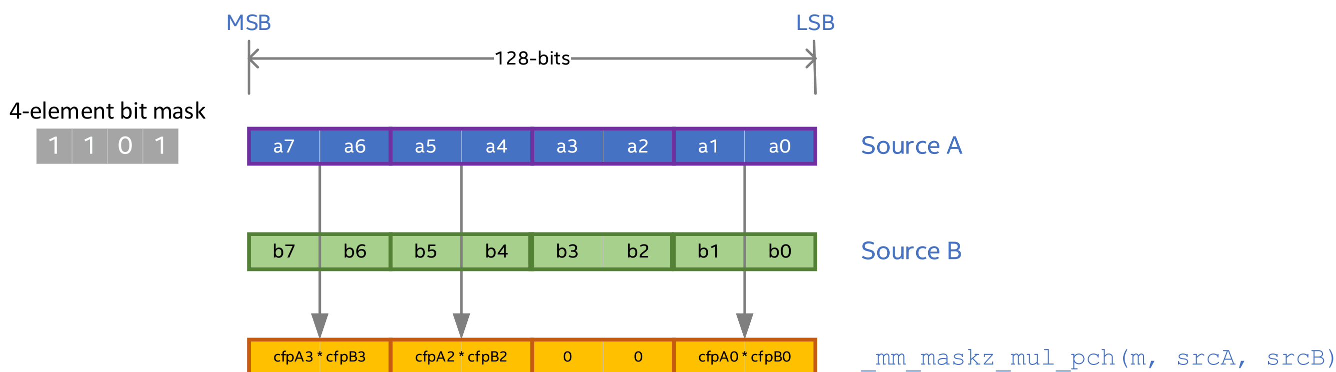

Figure 3. Illustration of a Masked Complex Multiplication

No direct numeric support is provided for complex operations such as addition, subtraction, and real-valued scaling, but their standard real-valued equivalent instructions can be used instead. However, if such an operation has to be masked on a per-complex-element basis, then the incoming complex-valued mask needs to be expanded into pairs of identical bits, one pair per

complex-element. An example of this is illustrated in Figure 4. Note that the incoming mask bit, which is per CFP16 element, needs to be expanded to duplicate each bit for the real-valued intrinsic.

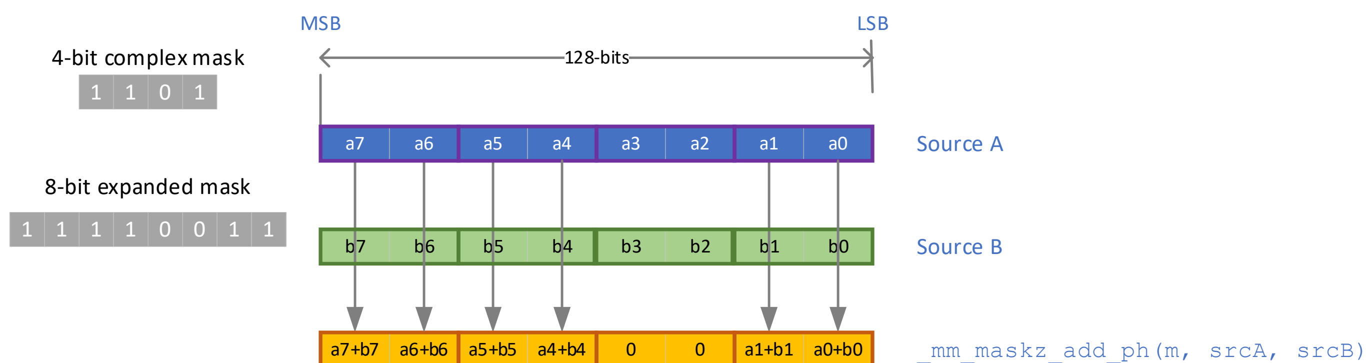

Figure 4. Illustration of Using a Real-Valued FP16 Vector Operation for Implementing a Masked Complex Addition

The operation to expand the incoming complex-mask-bits to generate real-valued mask could be performed in numerous different ways, but one efficient way to achieve this operation is shown in Figure 5. This code fragment uses the fast mask-to-vector and vector-to-mask instructions to effect the upscaling of the bit-mask elements.

```
__mmask8 getRealMaskFromComplexMask(__mmask8 m)
{
  // 4 incoming bits representing the 4 complex elements in a 128-bit register.
  // Each mask bit is converted into an entire element in a vector register
  // where a 0-mask generates 32x0, and a 1-mask generates 32x1. For example
  // 0010 -> [000....00], [000...000],  [111....111], [000....000]
  auto wholeElements = _mm_movm_epi32(m);

  // Each complex element can now be treated as a pair of 16-bit elements instead,
  // and the MSB of each 16-bit unit can extracted as a mask bit in its own right.
  return _mm_movepi16_mask(wholeElements);
}
```

Figure 5. Illustration of Using a Real-Valued FP16 Vector Operation for Implementing a Masked Complex Addition

It may also be necessary to perform a similar operation in reverse, where pairs of bits representing adjacent FP16 values need to be reduced in some way into a single bit representing the complete complex element (such as AND, OR). For example, if two complex vectors must be compared for equality then the individual FP16 elements must be compared for equality first. Then, if two adjacent mask bits are both set (that is, the logical AND of those bits), the complex element as a whole must be equal. This comparison test is illustrated in Figure 6. Note that some of the sub-elements in each CFP16 do compare equal when using the `_mm_cmp_ph_mask` intrinsic, but both elements in each CFP16 value must be equal for the complex values to be truly equal to each other.

Figure 6. Comparison Operation Between Two Complex-Valued Vectors. The Mask Bits Are Generated Using a Real-Valued Comparison, and Then Adjacent Bits Combined Using AND

One implementation of the function to combine adjacent mask bits using an AND operation is shown in Figure 7. Like the example above, it uses the mask-to-vector and vector-to-mask instructions to good effect.

```
__mmask8 getComplexMaskFromRealMask_AND(__mmask8 m)
{
  // 8 incoming bits representing the 4 real-valued elements in a 128-bit register.
  // Broadcast the bits into 8-bit elements of all 1's or all 0's.
  auto wholeElements = _mm_movm_epi8(m);

  // Generate an entire vector of 1's (typically a ternlogic will be used, which is
  // very cheap and can be done in parallel with the movm above, or hoisted when
  // used repeatedly.
  const auto allOnes = _mm_set1_epi16(-1);

  // Extract single mask bits from each 16-bit element which are the logical ANDs of the
  // MSBs of each incoming 8-bit element. Because the movm above generated all 0/1 bits
  // across the whole element the only combinations of values in each 32-bit unit are
  // both all zero, both all one, or one of each. The logical AND of the MSBs can only
  // occur when both 8-bit sub-elements are all ones, so this is equivalent to
  // comparing the 16-bit block as though it were entirely 1, which is a direct
  // equality comparison.
  return  _mm_cmp_epi16_mask(wholeElements, allOnes, _MM_CMPINT_EQ);
}
```

Figure 7. Function for Converting from a Real-Valued Mask to a Complex-Valued Mask By AND-Combining Adjacent Bits

Note that the individual mask bits are expanded to 8-bit elements and then compared for equality as 16-bit elements to combine adjacent elements. There is no need to expand to the same size as the data being processed (that is, 16/32-bit respectively in this case), since the bitwise pairing is independent of the original data element sizes. By using smaller registers, efficiency is very slightly improved compared to using wider registers.

The adjacent mask bits could also be combined using an OR operation, which might be useful if testing whether a complex value is NaN (that is, a complex value is NaN if either of its individual elements is NaN). A sequence for determining an OR of adjacent mask bits is shown in Figure 8.

```
__mmask8 getComplexMaskFromRealMask_OR(__mmask8 m)
{
  auto wholeElements = _mm_movm_epi16(m);

  // Similar logic to the AND variant above but now any 32-bit element which
  // isn't zero represents the logical OR or two adjacent 16-bit block
  // elements in one 32-bit block.
  return  _mm_cmp_epi32_mask(wholeElements, __m128i(), _MM_CMPINT_NE);
}
```

Figure 8. Function for Converting from a Real-Valued Mask To a Complex-Valued Mask By OR-Combining Adjacent Bits

4 Numerics

Using FP16 instead of the more conventional and widely used FP32 and FP64 formats introduces a number of interesting numeric behaviors. It is beyond the scope of this paper to discuss these fully or to describe the numeric methods required to build FP16 algorithms, but in this section, we highlight a few of the properties and behaviors of the FP16 number format and the consequences that arise from this.

4.1 Introduction to FP16 Number Format

An FP16 floating-point number is represented using 16 bits, which are laid out as shown in Figure 9. The figure also shows two other floating-point number formats for comparison, one being the common 32-bit FP32 format, and the other being the alternative 16-bit floating-point format called brain-float 16, which is used for machine learning. Note how bfloat16 is simply the upper 16-bits of the FP32 format, giving it the same dynamic range as FP32 but with considerably reduced precision, making this ideal for machine-learning applications. In contrast, the IEEE FP16 format modifies the sizes of the significand and the exponent to produce a more balanced blend of precision and range, which is more suitable for general purpose algorithms.

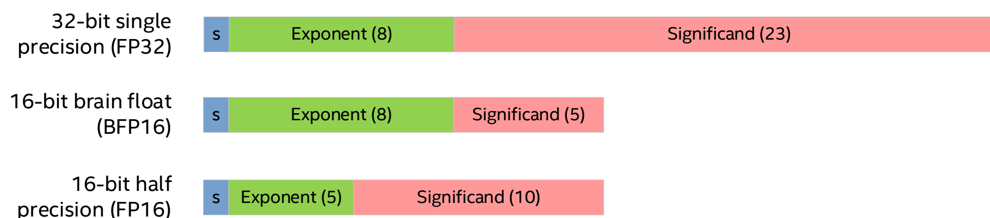

Figure 9. Bit Layout of Three Types of Floating-Point Formats

Certain bitwise operations can be used to manipulate the floating-point numbers without requiring special hardware support. For example, an absolute operation (that is, convert the value to its positive equivalent) can be implemented as a bitwise AND of the lower 15 bits of the value, thereby stripping off any sign bits. Similarly, functions like negate, negative-absolute (nabs), copy-sign, test-sign, and so on can also be implemented using existing Intel AVX-512 bitwise intrinsics.

4.2 Observations on Representing Numbers in FP16 Format

Although FP16 behaves functionally the same way as FP32 and FP64, the limited number of bits in its representations means that some surprising limits are imposed on the permitted values. In FP32 and FP64, most of the useful human-comprehensible numbers can be easily represented without considering too much about the limitations in value representation imposed by the floating-point format, but those limitations show up in FP16 limitations more easily. In Figure 10 some landmark values on the real-valued positive number line are illustrated, and in Table 6 further useful numbers are listed.

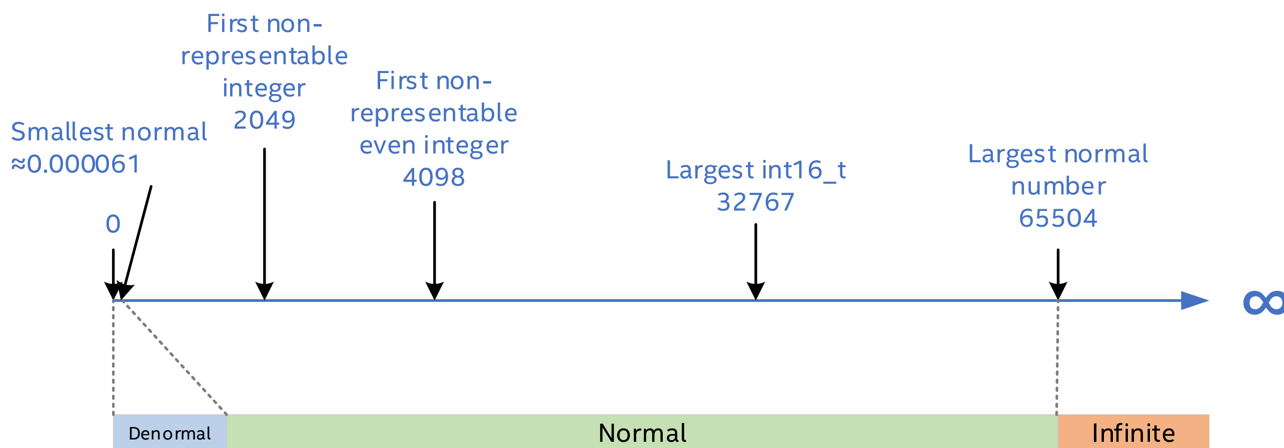

Figure 10. Landmark Numbers on the Real-Valued FP16 Axis

Table 6. Useful or Interesting FP16 Numbers

Value	Hex Representation	Description
0	`0x0000`	Zero
0.000000059604645	`0x0001`	Smallest denormal value
0.000060975552	`0x03ff`	Largest denormal value
0.00006103515625	`0x0400`	Smallest normal value
1	`0x3c00`	One
Inf	`0x7c00`	Positive Infinity
-Inf	`0xfc00`	Negative Infinity

Some consequences of using the FP16 number format include:

- Denormal numbers are not very small – it is easy to generate a number that gets too small to be represented using an FP16 normal value.
- Infinity starts very low down; it is only slightly less than the maximum value of an unsigned 16-bit integer. Overflow of values converted from 16-bit integers can quickly lead to infinities.
- Only integers up to a magnitude of 2048 can be fully represented. Beyond that, the permitted integers become very sparse very quickly. Rounding from a real integer type to an FP16 value introduces large absolute integer errors if the integer is above 2048.

These limitations may seem cumbersome at first, but there are good reasons why FP16 representation is a good fit for many signal-processing applications. Firstly, it is important to consider the Signal-to-Interference-plus-Noise ratio, or SINR. In a typical signal-processing system, such as a wireless receiver, the signals of interest are almost always subject to measurement noise. In the case of a wireless system, this noise would be introduced by receiver thermal-noise or in-band interference.

With FP16 number representation, any value on the real number-line within the normal range is subject to approximately -73.7 dB of quantization noise when quantized to FP16 format (IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—II: EXPRESS BRIEFS, VOL. 63, NO. 6, JUNE 2016: Quantization Noise Power Estimation for Floating-Point DSP Circuits Gabriel Caffarena, Senior Member, IEEE, and Daniel Menard, Member, IEEE[2]). To use common signal-processing parlance, the SINR is always ~73.7 dB – meaning that the quantizing error variance is $10^{-7.37}$ times the squared-magnitude of the signal. When compared to a typical received SINR of 25 dB, this means that the variance of the additional error introduced by the FP16 quantization is $10^{48.7/10}{\sim}74{,}131$ times lower than the measurement noise of the signal. In effect, it adds a negligible extra noise power.

Other signal processing requirements include dynamic range. The FP16 representation is able to maintain the 73.7 dB SINR over the complete dynamic range of a perfect 16-bit ADC. Care must be taken to exploit the floating-point aspect of FP16 and not directly convert integers to FP16, as squaring operations will likely result in “Inf”. However, this is easily overcome by converting 16-bit integers to FP16 and then scaling by a fixed constant: $1/256$ is a good choice.

[2] https://ieeexplore.ieee.org/document/7407669

4.3 Numeric Accuracy Guarantees

In any floating-point calculation, it is impossible to give the result of every possible computation because not every value has a valid representation. The output has to be quantized to a nearby value which *can* be represented in that number format. If the result is not representable, the distance between the next lowest representable value and the next highest representable value is called the unit-in-last-place, or ULP (and less commonly but equivalently, the unit-of-least-precision), and the actual answer will lie somewhere between the two. When the output value is rounded up or down to the nearest representable value, it therefore follows that the error in that calculation is no more than 0.5 ULP.

In IEEE 754 floating-point arithmetic, the standard mandates that the result of any hardware implementation will generate a correctly rounded result that has no more than 0.5 ULP of error when rounding `to nearest' for the following operations:

- Addition
- Subtraction
- Multiplication
- Division
- Square root
- Fused multiply-add

When rounding up, down, or toward zero, the error is less than 1 ULP.

The fused multiply-add is interesting because it guarantees that the intermediate result of the multiplication is kept in a higher precision form internally before being added. This means that the result of an FMA operation can have less overall error than doing a sequence of individual multiply and add instructions.

AVX512-FP16 is compliant with IEEE Standard 754-2019, and arithmetic operations on it are implemented in the spirit of the Standard (which does not require arithmetic operations for binary16). Consequently, all the operations listed above yield correctly rounded results. FP16 also contains a few instructions (not defined in IEEE 754-2019) that produce approximate results to within 0.5625 ULP error. These include:

- Reciprocal (rcp)
- Reciprocal square-root (rsqrt)

Further examination of these special cases is given in later sections.

Note also that complex multiplications (and fused multiplications) have an intermediate quantization to FP16 because, as described earlier, the hardware implements these operations as a sequence of FMAs. Each step of that sequence introduces quantizing, so the overall effect of the complete complex multiply has some small error.

4.4 Handling Denormal Values

FP16 differs from FP32 and FP64 floating-point by allowing computations involving denormals to be performed with no impact on cycle count. This is in contrast to FP32 and FP64 computation modes where handling denormals under some conditions can introduce cycle performance penalties. In FP32 and FP64 computations, when a denormal value is encountered the instruction might trap and call into a software routine to handle the computation instead, which increases the number of processor cycles required. Describing when this occurs and what the penalties would be is beyond the scope of this document. However, it is common to attempt to avoid denormal values in FP32 and FP64 computation where possible by modifying two FP execution flags:

DAZ	Denormals Are Zero	Any inputs that would be denormal are replaced by zero before use
FTZ	Flush To Zero	Any outputs that would be denormal are replaced by zero

Since FP16 handles denormals at full speed, all FP16 computations ignore the DAZ and FTZ flags and modifying these flags has no impact on FP16 numeric behavior or performance.

4.5 Embedded Rounding

In common with other AVX-512 instructions, FP16 allows the use of an instruction attribute called Static Rounding Mode. Rather than depending upon the contents of a global control register (called MXCSR) to set the floating-point rounding mode, most instructions in AVX512-FP16 can override the rounding mode behavior for only that instruction. Some permitted rounding modes are shown in Table 7.

For convenience, intrinsics are provided to give easy access to embedded rounding modes. For example, the FP16 addition instruction can have the rounding mode controlled by using the following intrinsic:

```
__m128h _mm512_add_round_ph (__m128h a, __m128h b, int rounding);
```

The third parameter, which supplies the rounding mode immediate, can be taken from the third column of Table 7, which describes four of the IEEE rounding modes and the required selector to invoke that behavior.

Table 7. Rounding Modes

IEEE 754-2019 Rounding Mode	Description	C Intrinsic Constant Selector
`roundTiesToEven`	Round toward nearest floating point, with ties to even	`_MM_FROUND_TO_NEAREST_INT | _MM_FROUND_NO_EXC`
`roundTowardPositive`	Round toward negative infinity	`_MM_FROUND_TO_NEG_INF | _MM_FROUND_NO_EXC`
`roundTowardNegative`	Round toward positive infinity	`_MM_FROUND_TO_POS_INF | _MM_FROUND_NO_EXC`
`roundTowardZero`	Round toward zero	`_MM_FROUND_TO_ZERO | _MM_FROUND_NO_EXC`

The following points should be noted:

- When a rounding mode is explicitly used, then this implies that the `suppress-all-exceptions' flag is also set for that instruction. Therefore, an instruction that uses embedded rounding never raises a floating-point exception.
- The C intrinsic constant selector name `_MM_FROUND_TO_NEAREST_INT` is not ideal, but that name has been historically used for so long in all common compilers that it is difficult to change to something more meaningful.
- Embedded rounding is only permitted on full width AVX-512 intrinsics (e.g., __mm512_OP_round_pX) or scalar operations (e.g., _mm_OP_round_sX). It is not permitted on AVX-512 VL encoded 128-bit or 256-bit operations.

4.6 Legacy FP16 Data Type Conversion

Two older Intel instruction sets already supported FP16 values as a storage format and provided conversion instructions to and from other data types. For example:

`_mm_cvtph_ps`	Convert from FP16 to FP32
`_mm_cvtps_ph`	Convert from FP32 to FP16

These instructions were originally available in the FP16C ISA for 128-bit and 256-bit registers. The Intel AVX-512 FP16 ISA further extended these instructions to work with 512-bit registers, and also added the option to conditionally mask selected elements (for example, `_mm512_mask_cvtph_ps`).

These instructions do not have embedded broadcast modes. It is recommended that the newer conversion instructions described in the next section be used instead.

4.7 FP16 Conversions to and from Other Data Types

The Intel AVX-512 FP16 contains a comprehensive set of instructions that convert to and from most of the other supported data types, with and without rounding.

Conversions from FP16 to other data types take the following intrinsic forms:

`_mm_cvtph_epi16`	Convert from half-precision to 16-bit integer
`_mm_cvtph_epi64`	Convert from half-precision to 64-bit integer
`_mm_cvtxph_ps`	Convert from half-precision to FP32

Note that an extra `x` appears in some of the intrinsics to differentiate them from their older FP16C/Intel AVX-512 F ISA counterparts. Only instructions that could be confused with older instruction sets have an `x` in their name (for example, the int16 conversion only appears in Intel AVX-512 FP16 so it does not need to be disambiguated).

When an FP16 denormal value is converted to a higher-precision FP32 or FP64 value, the denormal is converted to a normal representation in the output format.

Although the older conversion instructions perform type conversion as expected, they do not support embedded broadcasts. It is recommended to use the newer instructions wherever possible to get some instruction encoding advantages.

Conversions to FP16 format from other data types all take the intrinsic form shown in the following examples:

`_mm_cvtepi16_ph`	Convert from 16-bit integer to half-precision
`_mm_cvtepi64_ph`	Convert from 64-bit integer to half-precision
`_mm_cvtxph_ph`	Convert from FP32 to half-precision

Note that some care must be taken when converting from higher precision types into FP16. For example, conversion from a signed 16-bit integer value to FP16 generates the equivalent integers in FP16, albeit with some small loss possibly (for example, integer values greater than 2048 may be quantized to a nearby integer, not the exact integer). However, a more serious issue is that values that are converted from full-range 16-bit unsigned integer format are converted into FP16 values, which are at the very upper end of the permitted FP16 number range. Almost any numeric operation on such values could lead to overflow and the generation of infinities. In such scenarios, it is beneficial to perform some scaling on the value after conversion, to bring the range of the new values into the middle of the FP16 number range, thereby making it more difficult to hit infinities or denormals through normal compute operations.

Note that some care must be taken when converting to and from integer types to FP16. In particular, it must be noted that not all values in the 16-bit signed integer range of -32768 to +32767 can be represented in FP16. There will be some quantization effects with values above 2048. As discussed in Section 4.2, this additional quantizing noise power is negligible in most signal processing applications. However, the 16-bit integer range includes numbers that become close to Inf in FP16 format (values above 65504 are Inf). To avoid potential problems when performing typical signal-processing tasks such as cross-correlations, which operate in $volts^2$, 16-bit integer values should be scaled after conversion to FP16. A typical scale would be $1/256$. In this scheme, 32768 would be converted to 128.00f16. Note that both $128.00^2 = 16384.00$ and $(2/256)^2 = 0.00006103515625$ are within the normal range of FP16 values. This means that most typical signal-processing operations can be performed with values mostly in the normal range (with $(1/256)^2$ just falling outside of the normal range). So, by the simple expedient of applying a fixed scaling, FP16 representation can be used to comfortably span the dynamic-range presented by 16-bit ADCs and DACs.

4.8 Approximation Instructions and Their Uses

In common with Intel AVX-512, the new FP16 instructions support a number of approximation functions, including reciprocal (rcp), and reciprocal-sqrt (rsqrt). Although many instructions in Intel AVX-512 FP16 are accurate to within 0.5 ULP, as guaranteed by IEEE 754, the approximation instructions give very slightly less accurate results, but these are still useful, especially when compared with their equivalents in FP32 and FP64.

In FP32 and FP64 the approximation instructions are quite rough (that is, have a very high ULP error) and can only be used as a substitute for full-precision operations if combined with one or two Newton-Raphson iterations to refine the initial approximation to a point where it becomes sufficiently accurate. However, in FP16 the approximation functions give results that are so close to their full precision results - within 0.5 ULP for 98% of the possible values and within 0.5625 ULP for the remaining 2% of values – that there is no need to add Newton-Raphson iterations. This makes the approximation instructions very useful. They give virtually the correct answer, but with substantial benefits in performance over their full-accuracy counterparts. The following sections examine each approximation instruction in more detail.

4.8.1 Approximate Reciprocal

The reciprocal instruction in Intel AVX-512 FP16 behaves almost identically to the equivalent code sequence implemented using a division of the constant 1.0. For example, consider the following two code fragments:

```
__m512h trueRcp = _mm512_div_ph(_mm512_set1_ph(1.0f16), x);    // #1
__m512h approxRcp = _mm512_rcp_ph(x);                          // #2
```

The first, true reciprocal-by-division is guaranteed to be within 0.5 ULP, assuming rounding to nearest-even is used, but it takes approximately 15 cycles in 128 or 256-bit mode, and 24 cycles in 512-bit mode. It has a throughput of one instruction every 8 cycles in 128/256 bits mode, or one every 16 cycles in 512 bits mode. In contrast, the approximate reciprocal instruction is within 0.5 ULP for 98% of the possible valid input values, and the remaining 2% of values are within 0.5625 ULP, but it has a latency of only 4 cycles (or 6 cycles in 512 bits mode), and a throughput of 1 cycle (or 1.5 cycles in 512-bit mode). This dramatic improvement in compute performance of rcp_ph for almost no difference in numeric performance makes it ideal whenever that particular use case is required. Only when there is an absolute requirement for IEEE floating-point behavior should the division sequence be used instead.

4.8.2 Approximate Division

The two code fragments below show how a division could be implemented:

```
__m512h trueDiv = _mm512_div_ph(lhs, rhs);                  // #1
__m512h approxRcp = _mm512_mul_ph(lhs, mm512_rcp_ph(rhs));   // #2
```

The first of these uses the actual division instruction and is accurate to within 0.5 ULP (that is, correctly rounded, regardless of rounding mode).

The second sequence implements division by multiplying by the reciprocal, where the reciprocal is computed using the approximate function. We have already seen in the section above that the reciprocal approximation is very good, and this means that performing division using this sequence also turns out to be very good for most FP16 values. To illustrate how good the

approximation of the divide is, consider the heat-map shown in Figure 11. It shows how the ULP error changes for all the possible values of divisors and dividends. The green areas show that the approximation sequence gives identical results to a real division operation when the dividend is large and the divisor is small, or when the divisor is large and the dividend is small. The yellow region shows the cases where both the dividend and divisor fall into the middle-range of FP16 values, which is where a numerically well-designed algorithm falls, and indicates that 98% of values are within 0.5625 ULP of being correct, and every possible combination of dividend and divisor is never less accurate than about 1.5 ULP. The only places where the approximate division breaks down is when the divisor is very small (i.e., the left-hand red strip corresponding to the denormals), or very large (that is, the right-hand red strip where the exponent is at, or close to the maximum).

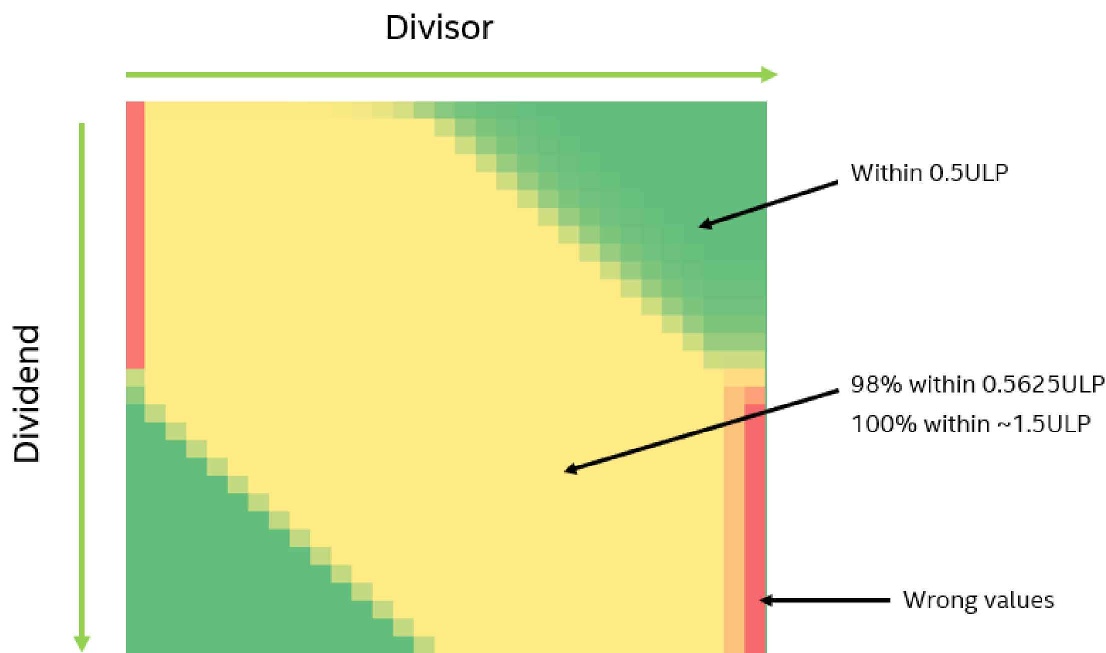

Figure 11. Heat Map Showing Relative ULP Error for Different Combinations of Divisor and Dividend Value Ranges

A division instruction is relatively expensive, taking 24 cycles with a throughput of 16 in 512-bit mode. In contrast, both multiply and reciprocal are cheap instructions, even when used in sequence, and consequently the approximation to division is ~3x faster. This speed, coupled with the low error for most FP16 values, means that well-designed algorithms could use the approximation sequence with little disadvantage.

4.8.3 Approximate Reciprocal Square Root

The reciprocal square root instruction in Intel AVX-512 FP16 is numerically very good. It gives a value that is within 0.5 ULP for 98% of the valid inputs, and the remaining 2% are within 0.5625 ULP of the true result.

An obvious implementation of a reciprocal square root, which uses the correctly rounded operations, is shown below:

```
__m512h rsqrtSequence = _mm512_div_ph(_mm512_set1_ph(1.0f16), _mm512_sqrt_ph(x));
```

This also gives a very good answer – within 0.5 ULP of the true result for 73% of possible inputs, and within 1 ULP for the remaining 23% - but that is slightly worse than the `_mm_rsqrt_ph` instruction itself, so the approximation intrinsic should be used in preference.

The approximate reciprocal square root takes 6 cycles, compared to the alternative sequence above that takes 48 cycles when in 512-bit mode.

4.9 Approximate Square Root

Although a square root instruction exists and is within 0.5 ULP of the true answer, it is possible to combine a product and rsqrt to get a good approximation, as shown in the code fragment below:

```
__m512h sqrtSequence = _mm512_mul_ph(x, _mm512_rsqrt_ph(x));
```

This sequence gives an answer that is identical to that from the sqrt instruction for 70% of the possible input values and is never more than 1 ULP away from the true result for any FP16 input value. The throughput of the product is twice the throughput of the reciprocal square root approximation, allowing for some flexibility in internal scheduling. The speed of this instruction sequence,

coupled with its negligible error, makes it suitable for fast sqrt for any algorithms except those that require guaranteed IEEE rounding.

5 Using Existing Intel AVX-512 Instructions to Augment FP16 Support

Intel AVX-512 FP16 provides purely numeric operations that require hardware support and cannot easily be implemented in any other way. For all other related operations needed to support the use of vector FP16 (for example, permuting FP16 elements in a register), it is necessary to use the instructions that are already provided as part of the existing Intel AVX-512 instruction set. As a convenience, the compiler provides many support functions. For example, the intrinsic called `_mm512_mask_blend_ph`, which blends FP16 elements from two registers into one, is implemented using the underlying mask_blend_epi16 instruction. In cases where the compiler does not provide such convenience functions it will be necessary for programmers to create a wrapper to handle this themselves. In this section we show how such functions could be implemented, list some of the common convenience instructions, and show one example where extra performance can be achieved by exploiting the Intel AVX-512 instruction set to handle floating-point comparisons more efficiently.

5.1 Using Existing Instructions to Extend Intel AVX-512 FP16 Intrinsics

Suppose we wish to use a bit mask to compress the elements of an FP16 vector register, creating something that would act as you would expect from a non-existent intrinsic called `_mm512_mask_compress_ph`. Although such an intrinsic does not exist, it can be created as shown in Figure 12.

```
__m512h compress_ph(__mmask32 mask, __m512h value)
{
  const auto asInt16 = _mm512_castph_si512(value);
  const auto comp = _mm512_mask_compress_epi16(mask, asInt16);
  return _mm512_castsi512_ph(comp);
}
```

Figure 12. Function to Implement the 16-Bit Compress Operation on FP16 Vector Elements

The strategy followed in the example is to take the incoming vector of FP16 values and recast to a vector of int16_t values using the castph_si512 intrinsic. The newly cast values are then processed as though they are a vector of int16_t elements instead. This step does not change the individual 16-bit element blocks; it just moves them around within the register. On completion, the value is recast back to its original type as a vector of FP16 elements.

Note that the cast operations have no runtime impact and are purely used to inform the compiler that the programmer is treating the underlying bits in the incoming register as though they were a different type. No type conversion takes place. In practice, the entire code sequence in the example function collapses into a single compress instruction.

This same strategy can be used to apply any sort of data movement instructions as a method of moving data around within FP16 vector registers. The code is somewhat verbose but can be easily hidden away as a library function. Furthermore, many common utility intrinsics of this sort have already been implemented in the Intel OneAPI compiler and can be used directly. It should only be necessary to build additional intrinsic support functions for more unusual operations.

In addition to data movement instructions, other bitwise operations like abs, nabs, negate, and copy-sign, can also be implemented using the underlying Intel AVX-512 foundation instructions.

5.2 Common Convenience Intrinsics

Convenience instructions are provided for the common cases where FP16 support is implemented with existing Intel AVX-512 instructions, without requiring the definition of verbose intrinsics given in the compress_ph example above. Table 8 provides a list of some common convenience intrinsics.

Table 8. Convenience Intrinsics

Mode	Purpose and Implementation
`_mm512_conj_pch`	Compute the conjugate of a complex number by using bitwise XOR operation to flip the sign bit of the imaginary elements.
`_mm512_abs_ph`	Compute the absolute numeric value of an FP16 element by using a bitwise AND instruction to mask off the sign bit.
`_mm512_mask_blend_ph`	Use the underlying `_mm512_mask_blend_epi16` intrinsic to provide an FP16 (_ph) equivalent.
`_mm512_permute[x]var_ph`	Reorder FP16 elements from one or two source registers.

Mode	Purpose and Implementation
`_mm512_reduce_[add/min/max]_ph`	Generate a sequence of instructions that performs a reduction operation across all the elements of an FP16 vector register. This is more complicated than the other examples because it performs a sequence of permutes and reorders to pull the data together and intersperses those operations with numeric reduction operations that perform addition, multiplication, minimum, and so on.

By using the int16_t minimum instead, the instruction takes only one cycle to execute, which is faster than the equivalent FP16 minimum, and can be used to accelerate latency or dependency-sensitive code. Note however that the throughput is lower than the equivalent FP16 minimum instruction, so code that is exclusively performing minimum operations may do better using the FP16 minimum.

For comparison operations all data types take the same number of cycles to compare so using the equivalent int16_t form of the instruction to perform a comparison makes no difference to performance.

6 Using FP16 Without Hardware Support

This document has been written with the intention that any software development with FP16 will be done on a 4th Gen Intel® Xeon® Scalable processor or better target processor where native support for the data format is available. However, it may be useful to be able to develop software on targets that do not have native hardware support, but where the software should still behave numerically in the same way as with native support. For example, it may be convenient to develop code on a laptop with an older instruction set. In that case, the software under development should be made to behave in the same numeric way that it would on a native machine. FP16 has numeric behavior that can be much more restrictive than the other floating-point formats. Care must be taken to ensure those behaviors are accurately represented and monitored during development.

In this section we discuss two ways to model the numeric behavior of FP16 software without access to native hardware: using emulation of the native 4th Gen Intel® Xeon® Scalable processor instruction set and using the compiler to generate equivalent software using older instructions.

6.1 Emulation of Targets with the FP16 Instruction Set

An Intel Xeon processor with FP16 support (or any other processor from Intel) can be emulated using the Intel® Software Development Emulator (Intel® SDE). In emulation, software can be compiled for a target processor, and the resulting binary can be executed using the emulator on a processor with an entirely different instruction set. Emulation works by rewriting the incoming instruction set on-the-fly to use whatever instructions are available on the host processor. The result of the emulation is bit-accurate with respect to a real processor with native support. Emulation is typically much slower than native execution, and emulation cannot be used for performance testing, but it is an excellent way to perform functional testing. The binary generated by the compiler can be the exact file that will eventually run on the real system, so the programmer can be confident that the compiled code will perform exactly as on the final system. This means that the source code behavior, compiler optimizations, or explicit use of intrinsics will be faithfully replicated in emulation, giving confidence that the developed software does what it is supposed to.

6.2 Compiler Synthesis of FP16 Instructions

Recent versions of LLVM, GCC, and oneAPI compilers may be used to generate FP16 code that is synthesized from older instruction sets[3]. For example, if the user explicitly uses `_Float16` data types, the compiler can still generate code that runs on processors that do not have native support by converting to and from other supported floating-point types as necessary. The degree to which the numeric results of such synthesized code match that of native hardware can vary somewhat, though, and some explanation of floating-point behaviors is required to understand the implications.

6.2.1 Intermediate (or Excess) Precision

Consider the following code snippet:

[3] The first versions of compilers that support FP16 synthesis are LLVM 15, GCC 12, and oneAPI 2023.1.

```
_Float16 intermediateMulAdd(_Float16 a, _Float16 b, _Float16 c, _Float16 d) {
  return a * b + c * d;
}

_Float16 explicitMulAdd(_Float16 a, _Float16 b, _Float16 c, _Float16 d)
{
  _Float16 t0 = a * b;
  _Float16 t1 = c * d;
  return t0 + t1;
}
```

In this example, two functions are computing the same basic mathematical operation, but because they are structured differently, they could produce different numeric answers depending upon what compiler options are in effect. This is because the first version has intermediate results that are never assigned to a variable. In such a case, the compiler is allowed to use a higher precision for the intermediate result. In the second fragment, the partial answers are stored back to a specific variable, which has the effect of fixing the precision of those partial results to a specific size. Whenever a variable is assigned or explicitly cast to another type, any higher precision value is quantized to its source precision.

Another cause of differences could be the use (or not) of the fused multiply-add (FMA) instruction when computing the final result. This potential difference in intermediate precision or use of FMA can also result in different numeric answers for the two functions.

In practice, when compiling for a processor that has native support for FP16 and that has an aggressive level of optimization enabled, the two fragments are likely to produce identical code. However, some compiler features that control the numeric behavior should be taken into consideration: the intermediate result evaluation level and the math optimization flags, including whether FMA is used.

6.2.2 Controlling the Intermediate Precision

LLVM/oneAPI[4] and GCC have slightly different behaviors when it comes to controlling intermediate precision.

LLVM and oneAPI support a command line option named `-ffp-eval-method`:

```
-ffp-eval-method=[source/double/extended]
```

The first option - `source` - specifies that whatever source type is used for the expression, it should be used for all intermediate results. In this example, since the variables are of type `_Float16`, the intermediate results are explicitly quantized to `_Float16` as well. The compiler might use FP32 multiply and addition instructions to perform the synthesis, but at each stage, the FP32 value is quantized back to FP16. This is likely to result in a numeric answer that is the same as if native hardware is used.

The second option - `double` - forces the compiler to promote the intermediate results to double precision (i.e., FP64). The entire expression for the first function in the code snippet is computed in double precision, and only quantized to FP16 when assigned back to the return value. This increased precision results in a different numeric value to that computed by the second function or by either function if computed on native hardware.

The partial disassembly output below shows the example code fragment from above compiled with both options in the OneAPI compiler:

-ffp-eval-method=source	-ffp-eval-method=double
# First op – convert up to FP32, multiply # then down-convert vcvtph2ps %xmm2, %xmm2 vmulss %xmm1, %xmm2, %xmm1 vcvtps2ph $4, %xmm1, %xmm1 # Second op – convert up to FP32, add	# Upconvert input arguments to FP64 vcvtph2ps %xmm3, %xmm3 vcvtss2sd %xmm3, %xmm3, %xmm3 # Series of arithmetic operations in FP64 # at high-precision vmulsd %xmm3, %xmm0, %xmm0

[4] oneAPI is based upon LLVM and shares this support.

`# then down-convert`
`vcvtph2ps %xmm1, %xmm1`
`vaddss %xmm0, %xmm1, %xmm0`
`vcvtps2ph $4, %xmm0, %xmm0`
	`vfmadd231sd %xmm2, %xmm1, %xmm0`
	`# Down convert the high-precision result`
	`callq __truncdfhf2@PLT`

Note how the left-hand source implementation quantizes back to FP16 after every arithmetic step, which makes it behave like real hardware. The right-hand version combines several arithmetic steps into one sequence, which operates at high-precision and gives results that have less error.

The third option creates FP80 format intermediate results and is rarely, if ever, used now.

GCC does not have the same command line, but it has one that achieves a similar effect:

```
-fexcess-precision=[standard/fast/16]
```

The standard option honors all assignments and casts of values, and any higher precision values are quantized to their destination type. The fast option also allows intermediate results to be higher precision, but does not honor any assignments or casts if the compiler expects that performance can be improved by ignoring those operations. Instead, it may keep values in a higher precision than expected, so this option should be used with care as it may lead to code that behaves unpredictably. The final option – `16` - has been introduced explicitly to support FP16 floating-point types, and it forces intermediate results to be quantized back to FP16. This behaves somewhat like using LLVM's `-ffp-eval-method=source` command line option.

In all the compilers mentioned, the default option for these command lines can vary according to changes in target architecture, language mode, or compiler version. Therefore, the user should explicitly set the desired command line option to ensure consistent handling of intermediate precision across multiple compilation scenarios.

6.2.3 Fast Math Optimizations

All modern compilers allow fast math operations to be enabled at the expense of non-conformance with IEEE standards. For example, compilers are allowed to contract `a * b + c' into an FMA operation with high internal precision, regardless of what excess-precision or fp-evaluation mode is active. The compiler is also allowed to ignore the possibility of NaNs, or to refactor or reorder floating-point operations. A full description of these behaviors is beyond the scope of this document. However, it should be noted that these behaviors, combined with synthesized instruction sequences at different precisions, ultimately result in numeric results that are different from those on native hardware. This is because the synthesized output of FP16 instructions is optimized according to whatever options are in effect in the compiler at the time, which can result in unpredictable behavior. It is recommended that fast math optimizations are turned off if the developer wants repeatable numeric behavior.

6.2.4 Summary and Recommendations for FP16 Emulation or Synthesis

If the developer wishes to write FP16 software without having native hardware available and to be confident that the numerics in the developed software are correct with respect to how it will work on a machine with native support, we recommend the following:

- Use the FP16 compiler and programming environment to generate a binary targeting a platform with native hardware support (for example, 4th Gen Intel Xeon Scalable processor). Use the Software Development Emulator to execute such a binary. This can be slower than the second option below, but it allows the developer to be confident of the exact behavior.
- Use the compiler to synthesize FP16 code using older or alternative instruction sets. Explicitly switch on either source-evaluation or set the intermediate precision to FP16 using the command line switches described earlier (i.e., don't rely on the defaults being set to a particular option). Also, disable any other math optimizations that might impact floating-point evaluation.

7 Math Library Support

The math libraries provided with Intel® compilers offer full functionality for the float16 data type (`_Float16`). Compiler support for the float16 data type can be enabled with `-xsapphirerapids` (ICX compiler). Float16-specific optimizations are available for vectorized math library calls.

Scalar math library functionality is available in the LIBM. These functions have not yet been optimized for Intel AVX-512 FP16 and currently rely on existing float32 implementations. Scalar float16 function names use the `f16` suffix (for example, expf16, logf16, sinf16, cosf16).

At the default accuracy level (four ULP or better), most common functions in the short vector math library (SVML) are optimized to take full advantage of the new Intel AVX-512 FP16 instruction set. Higher accuracy versions (one ULP) are also available, and most have been optimized; however, the one ULP versions frequently rely on single precision computation to

achieve the required accuracy. It is thus expected that the four ULP implementations provide noticeably better performance. The compiler generates SVML calls as part of loop vectorization. Compiler intrinsics for SIMD float16 calls (for example, `_mm512_log_ph`) are also available.

The Intel AVX-512 FP16 instruction set includes several operations that support the efficient implementation of math libraries. These operations are extensions of Intel AVX-512 transcendental support instructions and include VGETEXP, VGETMANT, VSCALEF, VFPCLASS, VREDUCE, VRNDSCALE (in addition to VRCP, VRSQRT).

VGETEXP (get normalized exponent) and VGETMANT (get normalized mantissa) are used together in implementations of functions such as log(), pow(), cbrt(). Without these operations, denormal and special inputs would require treatment in a separate path (or, alternatively, a slower main path that treats all inputs correctly). Denormals are reasonably likely to occur as inputs to float16 SVML calls due to the narrow float16 format range. The relative frequency of special inputs also increases with a wider SIMD length (32 packed float16 inputs per 512-bit SIMD register), so it is especially helpful to avoid branches. As an example, VGETEXP and VGETMANT can be used to reduce the log() computation to log(x)=VGETEXP(x)*log(2) + log(VGETMANT(x,8)). VGETMANT with an immediate value of 8 returns the normalized mantissa (in the [1,2) range) for positive inputs, and QNaN_Indefinite for negative inputs (which helps with special case handling).

VSCALEF(a,b)=a*$2^{floor(b)}$ is used in exponential and power functions; other possible applications include software division. This operation helps with correct overflow and underflow treatment in the main path. It also includes support for special exp() cases, thus eliminating the need for branches or other fixup code for this function family.

VFPCLASS tests for multiple special case categories (sNaN, negative finite, denormal, -Infinity, +Infinity, -0, +0, qNAN). This helps when redirecting special inputs to a secondary path (for example, in the pow() function), or to generate a fixup mask for setting special case results in the main path.

VRNDSCALE (round to specified number of fractional bits, using specified rounding mode) is used in function argument reduction, and to help generate lookup table indices (also as part of argument reduction). VRNDSCALE is a generalized form of round-to-integral, so it provides ceil/floor/trunc functionality, and also helps with floating-point remainder operations.

VREDUCE is closely related to VRNDSCALE: VREDUCE(x, imm) = x – VRNDSCALE(x,imm). This instruction helps further speed up argument reduction for certain functions (for example, exp2, pow).

The existing Intel AVX-512 permute operations (VPERM, VPERMT2, VPERMI2 for 16-bit and 32-bit data) provide fast vector gather support for those implementations that need lookup tables (up to 32 16-bit entries for VPERMW, up to 64 16-bit entries for VPERMT2W/VPERMI2Wn operations).

8 Summary

In this document, we have introduced the new Intel AVX-512 FP16 instruction set, discussed what changes it brings, and shown how to use it effectively. This document will continue to be updated to clarify behaviors, and to introduce new best practices as they become known.

If any information in this document appears to be missing or confusing, or could benefit from more explanation, contact your Intel representative.

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Intel technologies may require enabled hardware, software or service activation.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

The products described 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.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

0324/DN/WIT/PDF 669773-004US