Intel Intrinsics — AVX & AVX2 Learning Notes

August 15, 2018 · #SIMD, #Intel · 1159 words · 6 min

▼ Table of Contents

> AVX / AVX2 example code have been finished! Check it out here 🙂

Fundamentals of AVX Programming

Data Types

Data Type	Description
`__m128`	128-bit vector containing 4 `float` s
`__m128d`	128-bit vector containing 2 `double` s
`__m128i`	128-bit vector containing integers
`__m256`	256-bit vector containing 8 `float` s
`__m256d`	256-bit vector containing 4 `double` s
`__m256i`	256-bit vector containing integers

> - Each type starts with two underscores, an `m`, and the width of the vector in bits.
> - If a vector type ends in `d`, it contains `double` s, and if it doesn't have a suffix, it contains `float` s.
> - An integer vector type can contain any type of integer, from `char` s to `short` s to `unsigned long long` s. That is, an `_m256i` may contain 32 `char` s, 16 `short` s, 8 `int` s, or 4 `long` s. These integers can be signed or unsigned.

Function Naming Conventions

`_mm<bit_width>_<name>_<data_type>`

> - `<bit_width>` identifies the size of the vector returned by the function. For 128-bit vectors, this is empty. For 256-bit vectors, this is set to `256`.
> - `<name>` describes the operation performed by the intrinsic
> - `<data_type>` identifies the data type of the function's primary arguments
>
> > - `ps` - vectors contain `float` s (`ps` stands for packed single-precision)
> > - `pd` - vectors contain `double` s (`pd` stands for packed double-precision)
> > - `epi8/epi16/epi32/epi64` - vectors contain 8-bit/16-bit/32-bit/64-bit signed integers

> - `epu8/epu16/epu32/epu64` - vectors contain 8-bit/16-bit/32-bit/64-bit unsigned integers
> - `si128` / `si256` - unspecified 128-bit vector or 256-bit vector
> - `m128/m128i/m128d/m256/m256i/m256d` - identifies input vector types when they're different than the type of the returned vector

> A data type represents **m**emory and a function represents a **m**ulti**m**edia operation, so the AVX data types start with **two underscores** with **an** `m`, AVX functions start with **an underscore** with **two** `m`s.

Initialization Intrinsics

Initialization with Scalar Values

Function	Description
`_mm256_setzero_ps/pd`	Returns a floating-point vector filled with zeros
`_mm256_setzero_si256`	Returns an integer vector whose bytes are set to zero
`_mm256_set1_ps/pd`	Fill a vector with a floating-point value
`_mm256_set1_epi8/epi16/epi32/epi64x`	Fill a vector with an integer
`_mm256_set_ps/pd`	Initialize a vector with eight floats (ps)or four doubles (pd)
`_mm256_set_epi8/epi16/epi32/epi64x`	Initialize a vector with integers
`_mm256_set_m128/m128d/m128i`	Initialize a 256-bit vector with two 128-bit vectors
`_mm256_setr_ps/pd`	Initialize a vector with eight floats (ps) or four doubles (pd) in reverse order

Function	Description
`_mm256_setr_epi8/epi16/epi32/epi64x`	Initialize a vector with integers in reverse order

Loading Data from Memory

Data Type	Description
`_mm256_load_ps/pd`	Loads a floating-point vector from an aligned memory address
`_mm256_load_si256`	Loads an integer vector from an aligned memory address
`_mm256_loadu_ps/pd`	Loads a floating-point vector from an unaligned memory address
`_mm256_loadu_si256`	Loads an integer vector from an unalignedmemory address
`_mm_maskload_ps/pd` `_mm256_maskload_ps/pd`	Load portions of a 128-bit/256-bitfloating-point vector according to a mask
`(2)_mm_maskload_epi32/64` `(2)_mm256_maskload_epi32/64`	Load portions of a 128-bit/256-bitinteger vector according to a mask

> The last two functions are preceded with `(2)` because they're provided by AVX2, not AVX.

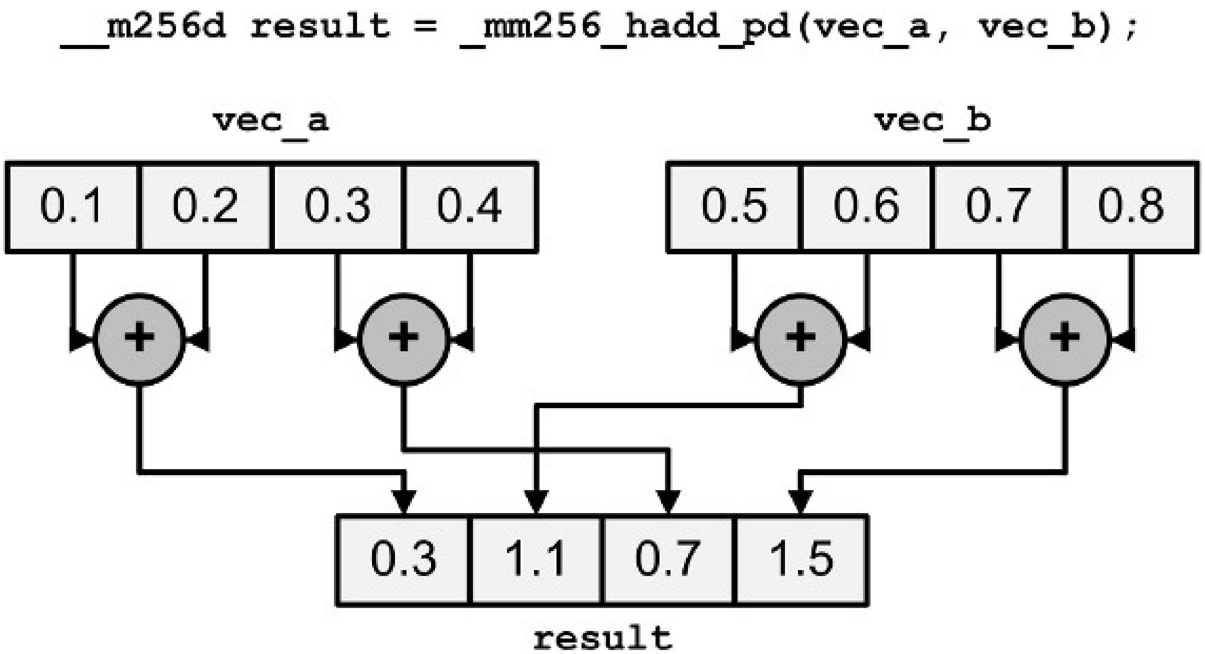
> Each `_mm256_load_*` intrinsic accepts a memory address that **must be aligned** on a 32-byte boundary.

Arithmetic Intrinsics

Addition and Subtraction

Data Type	Description
`_mm256_add_ps/pd`	Add two floating-point vectors
`_mm256_sub_ps/pd`	Subtract two floating-point vectors
`(2)_mm256_add_epi8/16/32/64`	Add two integer vectors
`(2)_mm236_sub_epi8/16/32/64`	Subtract two integer vectors
`(2)_mm256_adds_epi8/16` `(2)_mm256_adds_epu8/16`	Add two integer vectors with saturation
`(2)_mm256_subs_epi8/16` `(2)_mm256_subs_epu8/16`	Subtract two integer vectors with saturation
`_mm256_hadd_ps/pd`	Add two floating-point vectors horizontally
`_mm256_hsub_ps/pd`	Subtract two floating-point vectors horizontally
`(2)_mm256_hadd_epi16/32`	Add two integer vectors horizontally
`(2)_mm256_hsub_epi16/32`	Subtract two integer vectors horizontally
`(2)_mm256_hadds_epi16`	Add two vectors containing shorts horizontally with saturation
`(2)_mm256_hsubs_epi16`	Subtract two vectors containing shorts horizontally with saturation
`_mm256_addsub_ps/pd`	Add and subtract two floating-point **vectors**

> Functions that take **saturation** into account clamp the result to the minimum/maximum value that can be stored. Functions without saturation ignore the memory issue when saturation occurs.

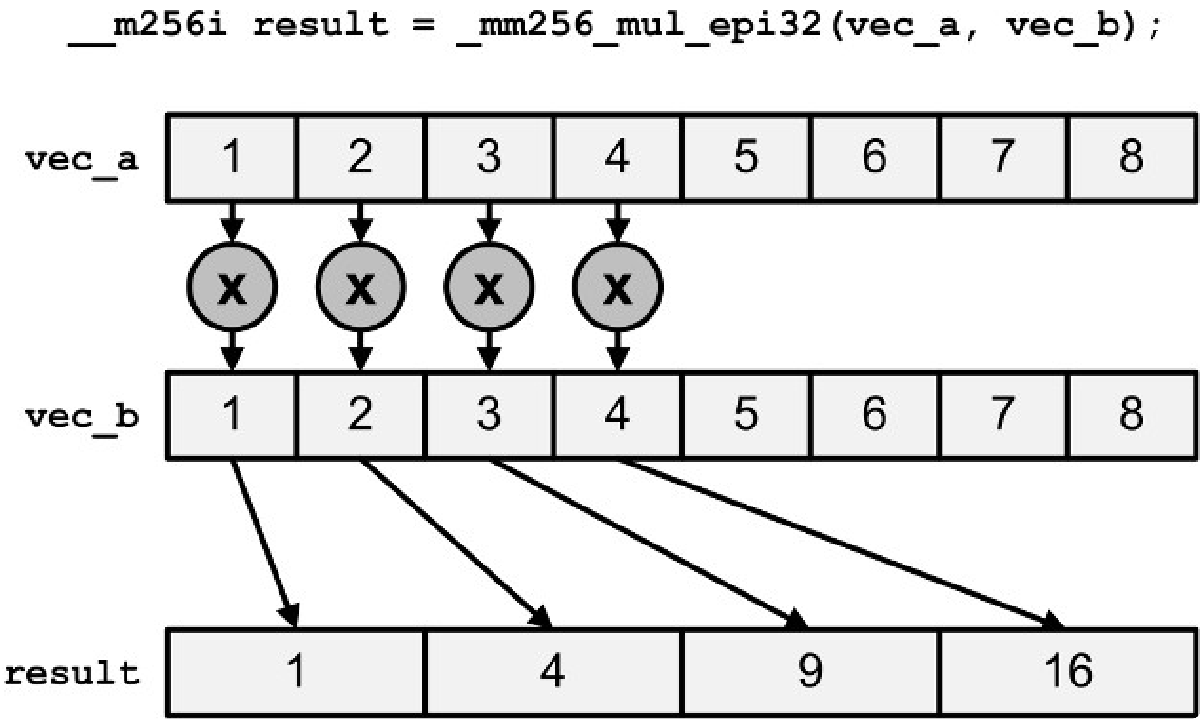


This may seem strange to add and subtract elements horizontally, but these operations are helpful when multiplying complex numbers.

`_mm256_addsub_ps/pd` , alternately subtracts and adds elements of two floating-point vectors. That is, **even elements are subtracted and odd elements are added** .

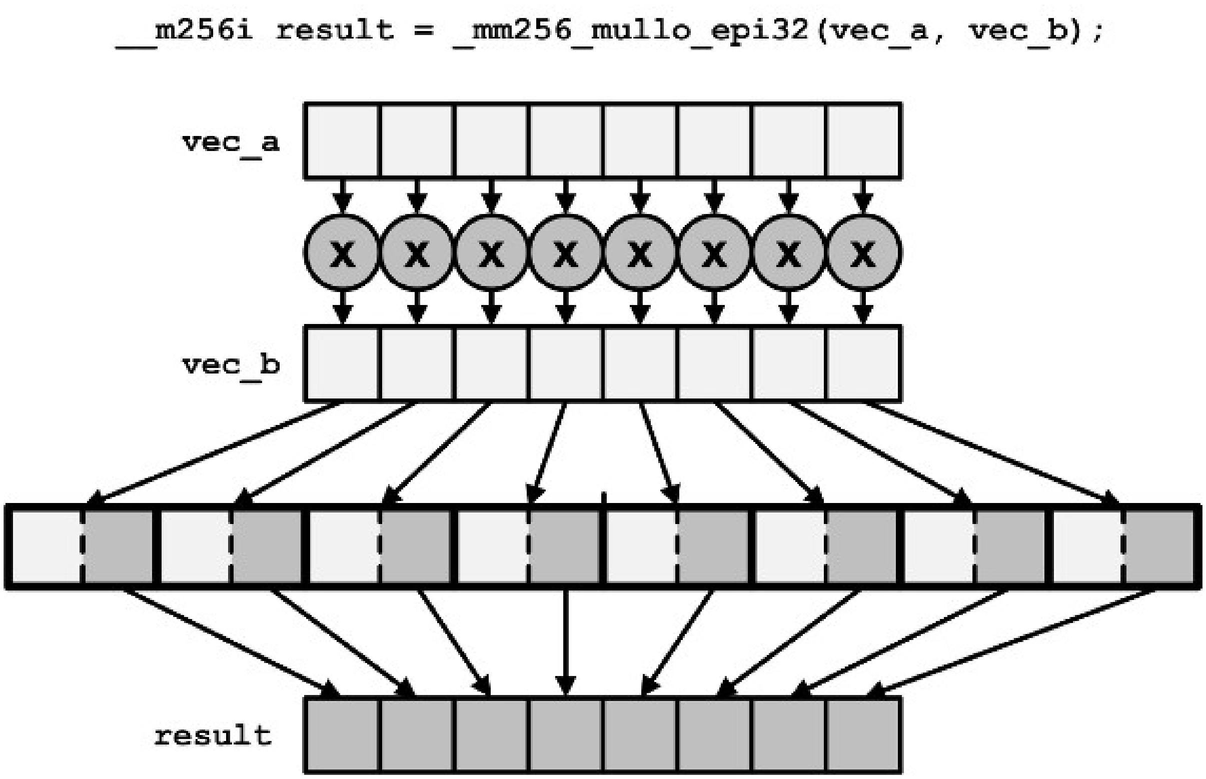
Multiplication and Division

Data Type	Description
`_mm256_mul_ps/pd`	Multiply two floating-point vectors
`(2)_mm256_mul_epi32` `(2)_mm256_mul_epu32`	Multiply the lowest four elements of vectors containing 32-bit integers
`(2)_mm256_mullo_epi16/32`	Multiply integers and store low halves
`(2)_mm256_mulhi_epi16` `(2)_mm256_mulhi_epu16`	Multiply integers and store high halves
`(2)_mm256_mulhrs_epi16`	Multiply 16-bit elements to form 32-bit elements
`_mm256_div_ps/pd`	Divide two floating-point vectors



> This image is **WRONG** !!!
>
> Please read the reference from [this manual].

> Only the four low elements of the `_mm256_mul_epi32` and `_mm256_mul_epu32` intrinsics are multiplied together, and the result is a vector containing four long integers.



> They multiply every element of both vectors store only the low half of each product

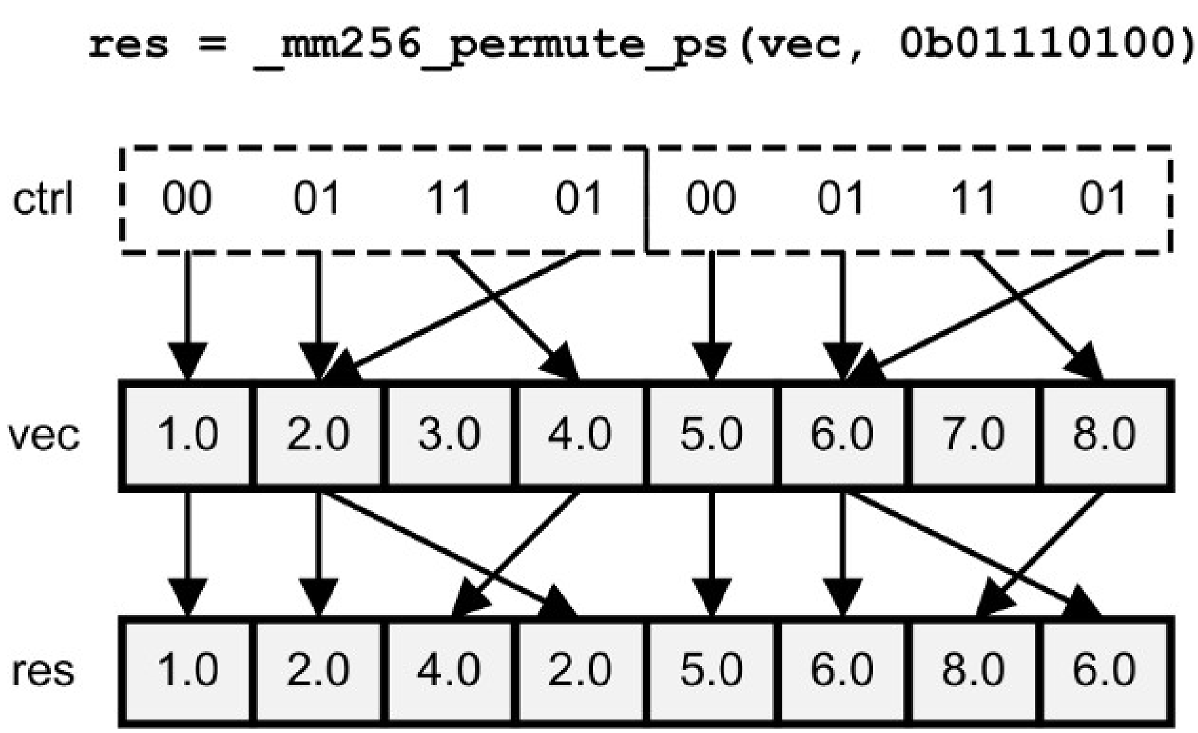
Fused Multiply and Add (FMA)

Data Type	Description
`(2)_mm_fmadd_ps/pd/` `(2)_mm256_fmadd_ps/pd`	Multiply two vectors and add the product to a third (res = a * b + c)
`(2)_mm_fmsub_ps/pd/` `(2)_mm256_fmsub_ps/pd`	Multiply two vectors and subtract a vector from the product (res = a * b - c)
`(2)_mm_fmadd_ss/sd`	Multiply and add the lowest element in the vectors (res[0] = a[0] * b[0] + c[0])
`(2)_mm_fmsub_ss/sd`	Multiply and subtract the lowest element in the vectors (res[0] = a[0] * b[0] - c[0])
`(2)_mm_fnmadd_ps/pd` `(2)_mm256_fnmadd_ps/pd`	Multiply two vectors and add the negated product to a third (res = -(a * b) + c)
`(2)_mm_fnmsub_ps/pd/` `(2)_mm256_fnmsub_ps/pd`	Multiply two vectors and add the negated product to a third (res = -(a * b) - c)
`(2)_mm_fnmadd_ss/sd`	Multiply the two lowest elements and add the negated product to the lowest element of the third vector (res[0] = -(a[0] * b[0]) + c[0])
`(2)_mm_fnmsub_ss/sd`	Multiply the lowest elements and subtract the lowest element of the third vector from the negated product (res[0] = -(a[0] * b[0]) - c[0])
`(2)_mm_fmaddsub_ps/pd/` `(2)_mm256_fmaddsub_ps/pd`	Multiply two vectors and alternately add and subtract from the product (res = a * b +/- c) (Odd add, even sub)
`(2)_mm_fmsubadd_ps/pd/` `(2)_mmf256_fmsubadd_ps/pd`	Multiply two vectors and alternately subtract and add from the product (res = a * b -/+ c) (Odd sub, even add)

Permuting and Shuffling

Permuting

Data Type	Description
`_mm_permute_ps/pd` `_mm256_permute_ps/pd`	Select elements from the input vector based on an 8-bit control value
`(2)_mm256_permute4x64_pd/` `(2)_mm256_permute4x64_epi64`	Select 64-bit elements from the input vector based on an 8-bit control value
`_mm256_permute2f128_ps/pd`	Select 128-bit chunks from two input vectors based on an 8-bit control value
`_mm256_permute2f128_si256`	Select 128-bit chunks from two input vectors based on an 8-bit control value
`_mm_permutevar_ps/pd` `_mm256_permutevar_ps/pd`	Select elements from the input vector based on bits in an integer vector
`(2)_mm256_permutevar8x32_ps` `(2)_mm256_permutevar8x32_epi32`	Select 32-bit elements (`float` s and `int` s) using indices in an integer vector



Shuffling

Data Type	Description
`_mm256_shuffle_ps/pd`	Select floating-point elements according to an 8-bit value
`_mm256_shuffle_epi8/ _mm256_shuffle_epi32`	Select integer elements according to an8-bit value
`(2)_mm256_shufflelo_epi16/ (2)_mm256_shufflehi_epi16`	Select 128-bit chunks from two input vectors based on an 8-bit control value

> For `_mm256_shuffle_pd` , only the high four bits of the control value are used. If the input vectors contain `int` s or `float` s, all the control bits are used. For `_mm256_shuffle_ps` , the first two pairs of bits select elements from the first vector and the second two pairs of bits select elements from the second vector.