Intrinsic Summary

4 avril 2019

Table des matières

1 SSE family

SSE - SSE2 – SSE3 – SSE4.1 – SSS4.2, and only using instructions related to integers no matter what integer → ___mmXXXi.
Need #include <emmintrin.h>

1.1 Set or eight

We can have the supplied values in reverse order with _mm_setr_epiX()

INTRINSIC	note
___m128i _mm_set_epi8(char $e_{15}, \ldots$, char e_0)	Set 16 packed 8-bit integers in dst
___m128i _mm_set_epi16 (short $e_7, \ldots$, short e_0)	Set 8 packed 16-bit integers in dst.
___m128i _mm_set_epi32 (int $e_3, \ldots$, int e_0)	Set 4 packed 32-bit integers in dst
___m128i _mm_set_epi64(___m64, ___m64)	Set 2 packed 64-bit integers in dst
___m128i _mm_set_epi64x(___int64 e1, ___int64 e2)	Set 2 packed 64-bit integers in dst, works with uint64
___m128i _mm_set1_epi8(char a)	Broadcast 8-bit integer a to all elements of dst
___m128i _mm_set1_epi16(short a)	Broadcast 16-bit integer a to all all elements of dst.
___m128i _mm_set1_epi32(int a)	Broadcast 32-bit integer a to all elements of dst.
___m128i _mm_set1_epi64(___m64 a)	Broadcast 64-bit integer a to all elements of dst.
___m128i _mm_set1_epi64x(___int64)	Broadcast 64-bit integer a to all elements of dst.
___m128i _mm_setzero_si128()	Return vector of type ___m128i with all elements set to zero.
___m128i _mm_cvtsi32_si128 (int a)	dst$[31:0] = a[31:0]$, dst$[127:32] = 0$
___m128i _mm_cvtsi64_si128(___int64 a)	dst$[63:0] = a[63:0]$, dst$[127:64] = 0$
___m128i _mm_cvtsi64_si128x (___int64 a)	dst$[63:0] = a[63:0]$, dst$[127:64] = 0$

FIGURE 1 – Set with SSE and 128bits vectors

1.2 Load

If ALIGNED -> on 16-byte

INTRINSIC	note
___m128i _mm_lddqu_si128(___m128i const* mem_addr)	Load 128-bitsfrom unaligned memory into dst.
___m128i _mm_load_si128(___m128i const* mem_addr)	Load 128-bits. Aligned on 16-byte
___m128i _mm_loadl_epi64(___m128i const* mem_addr)	copy 64bits in $[63:0]$ and set to 0 the rest
___m128i _mm_loadu_si128(___m128i const* mem_addr)	Load 128-bits. Unaligned
___m128i _mm_loadu_si16 (void const* mem_addr)	Load unaligned 16-bit into the first element of dst.
___m128i _mm_loadu_si32 (void const* mem_addr)	Load unaligned 32-bit into the first element of dst.
___m128i _mm_loadu_si64	Load unaligned 64-bit into the first element of dst.
___m128i _mm_stream_load_si128 (___m128i * mem_addr)	Load 128-bitsusing a non-temporal memory hint. Aligned

FIGURE 2 – load with SSE and 128bits vectors

1.3 Shift

INTRINSIC	note
___m128i _mm_bslli_si128 (___m128i a, int imm8)	Shift a left by imm8 bytes while shifting in zeros
___m128i _mm_bsrli_si128 (___m128i a, int imm8)	Shift a right by imm8 bytes while shifting in zeros
___m128i _mm_sll_epi16 (___m128i a, ___m128i count)	Shift 16-bit integers in a left by count while shifting in zeros
___m128i _mm_sll_epi32 (___m128i a, ___m128i count)	Shift 32-bit integers in a left by count while shifting in zeros
___m128i _mm_sll_epi64 (___m128i a, ___m128i count)	Shift 64-bit integers in a left by count while shifting in zeros
___m128i _mm_slli_epi16 (___m128i a, int imm8)	Shift 16-bit integers in a left by imm8 while shifting in zeros
___m128i _mm_slli_epi32 (___m128i a, int imm8)	Shift 32-bit integers in a left by imm8 while shifting in zeros
___m128i _mm_slli_epi64 (___m128i a, int imm8)	Shift 64-bit integers in a left by imm8 while shifting in zeros
___m128i _mm_slli_epi128 (___m128i a, int imm8)	Shift 128-bit integers in a left by imm8 while shifting in zeros

FIGURE 3 – shift with SSE and 128bits vectors

We have the same with ___m128i _mm_sra_epi16 (___m128i a, ___m128i count) where we shift a right by count while shifting in sign bits, and store the results in dst. sra goes from epi16 to epi32. srai (uses int) from epi16 to 32. srl (uses count) from epi16 to 64. srli (uses int) from epi16 to 128.

1.4 XOR OR AND

INTRINSIC	note
___m128i _mm_and_si128 (___m128i a, ___m128i b)	Compute the bitwise AND of a and b
___m128i _mm_andnot_si128 (___m128i a, ___m128i b)	Compute the bitwise NOT of a and then AND with b
___m128i _mm_xor_si128 (___m128i a, ___m128i b)	Compute the bitwise XOR of a and b
___m128i _mm_or_si128 (___m128i a, ___m128i b)	Compute the bitwise OR of a and b.

FIGURE 4 – XOR OR AND with SSE and 128bits vectors

1.5 Shuffle/table lookup

INTRINSIC	note
___m128i _mm_shuffle_epi32 (___m128i a, int imm8)	Shuffle 32-bit integers in a using the control in imm8
___m128i _mm_shuffle_epi8 (___m128i a,___m128i a)	Shuffle packed 8-bit integers in a according to shuffle control mask in the corresponding 8-bit element of b
___m128i _mm_shuffle_epi16 (___m128i a, int imm8)	Shuffle 16-bit integers in the high 64 bits of a using the control in imm8. Store the results in the high 64 bits of dst, with the low 64 bits being copied from a to dst.
___m128i _mm_shuffle_epi16 (___m128i a, int imm8)	Shuffle 16-bit integers in the low 64 bits of a using the control in imm8. Store the results in the low 64 bits of dst, with the high 64 bits being copied from a to dst.

FIGURE 5 – Shuffle/table lookup with SSE and 128bits vectors

1.6 byte interleaving

INTRINSIC	note
___m128i _mm_unpackhi_epi8 (___m128i a, ___m128i b)	Unpack and interleave 8-bit integers from the high half of a and b
___m128i _mm_unpackhi_epi16 (___m128i a, ___m128i b)	Unpack and interleave 16-bit integers from the high half of a and b
___m128i _mm_unpackhi_epi32 (___m128i a, ___m128i b)	Unpack and interleave 32-bit integers from the high half of a and b
___m128i _mm_unpackhi_epi64 (___m128i a, ___m128i b)	Unpack and interleave 64-bit integers from the high half of a and b

FIGURE 6 – Shuffle/table lookup with SSE and 128bits vectors

We have the exact same with ___m128i _mm_unpacklo_epi8 (___m128i a, ___m128i b) from epi8 to epi64.

2 AVX

2.1 Link SSE-AVX

___m128i _mm256_extractf128_si256 (___m256i a, const int imm8), Extract 128 bits (composed of integer data) from a, selected with imm8, and store the result in dst.

2.2 Set

INTRINSIC	note
___m256i _mm256_set_epi8 (char $e_{31}, \ldots,$ char e_0) ___m256i _mm256_set_epi16 (short $e_{15}, \ldots,$ short e_0)	Set packed 8-bit integers in dst Set packed 16-bit integers in dst
___m256i _mm256_set_epi32 (int $e_7, \ldots,$ int e_0)	Set packed 32-bit integers in dst
___m256i _mm256_set_epi64x (___int64 $e_3, \ldots,$ ___int64 e_0)	Set packed 32-bit integers in dst
___m256i _mm256_set_m128i (___m128i hi, ___m128i lo)	Set packed ___m128i vectors in dst
___m256i _mm_set1_epi8(char a)	Broadcast 8-bit integer a to all elements of dst
___m256i _mm_set1_epi16(short a)	Broadcast 16-bit integer a to all all elements of dst.
___m256i _mm_set1_epi32(int a)	Broadcast 32-bit integer a to all elements of dst.
___m256i _mm_set1_epi64x(long long a)	Broadcast 64-bit integer a to all elements of dst.
___m256i _mm256_setzero_si256 (void)	Return vector of type ___m256i with all elements set to 0

FIGURE 7 – SET with AVX and 256bits vectors

2.3 Load

Alignement is on 32bytes. lddqu might be slightly more efficient thant loadu when the data crosses a cache line boundary.

INTRINSIC	note
__m256i _mm256_lddqu_si256 (___m256i const * mem_addr)	Load 256-bits of unaligned memory i
___m256i _mm256_load_si256 (___m256i const * mem_addr)	Load 256-bits of aligned memory into
___m256i _mm256_load_si256 (___m256i const * mem_addr)	Load 256-bits of unaligned memory i
___m256i _mm256_loadu2_m128i (___m128i const* hiaddr, ___m128i const* loaddr)	Load two 128-bits unaligned and com
void _mm256_stream_si256 (___m256i * mem_addr,___m256i a)	Store 256-bits aligned using a non-te

FIGURE 8 – LOAD with AVX and 256bits vectors

2.4 Shift

INTRINSIC	note

FIGURE 9 – shift with AVX and 256bits vectors

2.5 XOR OR AND

INTRINSIC	note

FIGURE 10 – XOR OR AND with AVX and 256bits vectors

2.6 Shuffle/table lookup

INTRINSIC	note
___m256i _mm256_permute2f128_si256 (___m256i a, ___m256i b, int imm8)	Shuffle 128-bits selected by imm8 from a and

FIGURE 11 – Shuffle/table lookup with AVX and 256bits vectors

3 AVX2

3.1 Set

INTRINSIC	note
___m128i _mm_broadcastb_epi8 (___m128i a)	Broadcast the low packed 8-bit integer from a to all elements of dst
___m256i _mm256_broadcastb_epi8 (___m128i a)	Broadcast the low packed 8-bit integer from a to all elements of dst
___m128i _mm_broadcastd_epi32 (___m128i a)	Broadcast the low packed 32-bit integer from a to all elements of ds
___m256i _mm256_broadcastd_epi32 (___m128i a)	Broadcast the low packed 32-bit integer from a to all elements of ds
___m128i _mm_broadcastq_epi64 (___m128i a)	Broadcast the low packed 64-bit integer from a to all elements of ds
___m256i _mm256_broadcastq_epi64 (___m128i a)	Broadcast the low packed 64-bit integer from a to all elements of ds
___m256i _mm_broadcastsi128_si256 (___m128i a)	Broadcast 128 bits of integer data from a to all 128-bit lanes in dst.
___m256i _mm256_broadcastsi128_si256 (___m128i a)	Broadcast 128 bits of integer data from a to all 128-bit lanes in dst.
___m128i _mm_broadcastw_epi16 (___m128i a)	Broadcast the low packed 16-bit integer from a to all elements of ds
___m256i _mm256_broadcastw_epi16 (___m128i a)	Broadcast the low packed 16-bit integer from a to all elements of ds

FIGURE 12 – SET with AVX and 256bits vectors

3.2 Load

Alignement is on 32bytes. lddqu might be slightly more efficient thant loadu when the data crosses a cache line boundary.

INTRINSIC	note
___m128i _mm_maskload_epi32 (int const* mem_addr, ___m128i mask)	Load 32-bit using mask
___m256i _mm256_maskload_epi32 (int const* mem_addr, ___m256i mask)	Load 32-bitusing mask
___m128i _mm_maskload_epi64 (___int64 const* mem_addr, ___m128i mask)	Load 64-bit integers using mask
___m256i _mm256_maskload_epi64 (___int64 const* mem_addr, ___m256i mask)	Load 64-bit using mask
___m256i _mm256_stream_load_si256 (___m256i const* mem_addr)	Load 256-bits aligned using a non-tempo

FIGURE 13 – LOAD with AVX2 and 256bits vectors

3.3 Shift

INTRINSIC	note
___m256i _mm256_bslli_epi128 (___m256i a, const int imm8)	Shift 128-bit lanes left by imm8 bytes while shifting in zeros
___m256i _mm256_bsrli_epi128 (___m256i a, const int imm8)	Shift 128-bit lanes right by imm8 bytes while shifting in zero
___m256i _mm256_sll_epi16 (___m256i a, ___m128i count)	Shift 16-bit left by count while shifting in zeros
___m256i _mm256_sll_epi32 (___m256i a, ___m128i count)	Shift 32-bit left by count while shifting in zeros
___m256i _mm256_sll_epi64 (___m256i a, ___m128i count)	Shift 64-bit left by count while shifting in zeros
___m256i _mm256_slli_epi16 (___m256i a, int imm8)	Shift 16-bit left by imm8 while shifting in zeros
___m256i _mm256_slli_epi32 (___m256i a, int imm8)	Shift 32-bit left by imm8 while shifting in zeros
___m256i _mm256_slli_epi64 (___m256i a, int imm8)	Shift 64-bit left by imm8 while shifting in zeros
___m256i _mm256_slli_si256 (___m256i a, const int imm8)	Shift 128-bit lanes left by imm8 bytes while shifting in zeros
___m128i _mm_sllv_epi32 (___m128i a, ___m128i count)	Shift 32-bit left by the amount specified by the correspondin
___m256i _mm256_sllv_epi32 (___m256i a, ___m256i count)	Shift 32-bit left by the amount specified by the correspondin
___m128i _mm_sllv_epi64 (___m128i a, ___m128i count)	Shift 64-bit left by the amount specified by the correspondin
___m256i _mm256_sllv_epi64 (___m256i a, ___m256i count)	Shift 64-bit left by the amount specified by the correspondin
___m256i _mm256_sra_epi16 (___m256i a, ___m128i count)	Shift 16-bit right by count while shifting in sign bits
___m256i _mm256_sra_epi32 (___m256i a, ___m128i count)	Shift 32-bit right by count while shifting in sign bits
___m256i _mm256_srai_epi16 (___m256i a, int imm8)	Shift16-bit in a right by imm8 while shifting in sign bits
___m256i _mm256_srai_epi32 (___m256i a, int imm8)	Shift 32-bit in a right by imm8 while shifting in sign bits
___m128i _mm_srav_epi32 (___m128i a, ___m128i count)	Shift 32-bit right by the amount specified by the correspond
___m256i _mm256_srav_epi32 (___m256i a, ___m256i count)	Shift 32-bit integers right by the amount specified by the co
___m256i _mm256_srl_epi16 (___m256i a, ___m128i count)	Shift 16-bit right by count while shifting in zeros
___m256i _mm256_srl_epi32 (___m256i a, ___m128i count)	Shift 32-bit right by count while shifting in zeros
___m256i _mm256_srl_epi64 (___m256i a, ___m128i count)	Shift 64-bit right by count while shifting in zeros
___m256i _mm256_srli_epi16 (___m256i a, int imm8)	Shift 16-bit right by imm8 while shifting in zeros
___m256i _mm256_srli_epi32 (___m256i a, int imm8)	Shift 32-bit right by imm8 while shifting in zeros
___m256i _mm256_srli_epi64 (___m256i a, int imm8)	Shift 64-bit right by imm8 while shifting in zeros
___m256i _mm256_srli_epi256 (___m256i a, const int imm8)	Shift 18-bit lanes right by imm8 while shifting in zeros
___m128i _mm_srlv_epi32 (___m128i a, ___m128i count)	Shift 32-bit right by the amount specified by the correspond
___m256i _mm256_srlv_epi32 (___m256i a, ___m256i count)	Shift 32-bit right by the amount specified by the correspond
___m128i _mm_srlv_epi64 (___m128i a, ___m128i count)	Shift 64-bit right by the amount specified by the correspond
___m256i _mm256_srlv_epi64 (___m256i a, ___m256i count)	Shift 64-bit right by the amount specified by the correspond

FIGURE 14 – shift with AVX2 and 256bits vectors

3.4 XOR OR AND

INTRINSIC	note
___m256i _mm256_xor_si256 (___m256i a, ___m256i b)	Compute the bitwise XOR of 256 bits in a and b
___m256i _mm256_or_si256 (___m256i a, ___m256i b)	Compute the bitwise OR of 256 bits in a and b
___m256i _mm256_and_si256 (___m256i a, ___m256i b)	Compute the bitwise AND of 256 bits in a and b

FIGURE 15 – XOR OR AND with AVX and 256bits vectors

3.5 Shuffle/table lookup

INTRINSIC	note
___m256i _mm256_permute2f128_si256 (___m256i a, ___m256i b, int imm8)	Shuffle 128-bits selected by imm8 from a and
___m256i _mm256_shuffle_epi32 (___m256i a, const int imm8)	Shuffle 32-bit in a within 128-bit lanes using t
___m256i _mm256_shuffle_epi8 (___m256i a, ___m256i b)	Shuffle 8-bit a within 128-bit lanes according
___m256i _mm256_shufflehi_epi16 (___m256i a, const int imm8)	Shuffle 16-bit in the high 64 bits of 128-bit la
___m256i _mm256_shufflelo_epi16 (___m256i a, const int imm8)	Shuffle 16-bit in the low 64 bits of 128-bit lan

FIGURE 16 – Shuffle/table lookup with AVX and 256bits vectors

3.6 byte interleaving

INTRINSIC	note
___m256i _mm256_unpackhi_epi8 (___m256i a, ___m256i b)	Unpack and interleave 8-bit integers from the high half of ea
___m256i _mm256_unpackhi_epi16 (___m256i a, ___m256i b)	Unpack and interleave 16-bit integers from the high half of e
___m256i _mm256_unpackhi_epi32 (___m256i a, ___m256i b)	Unpack and interleave 32-bit integers from the high half of e
___m256i _mm256_unpackhi_epi64 (___m256i a, ___m256i b)	Unpack and interleave 64-bit integers from the high half of e
___m256i _mm256_unpacklo_epi8 (___m256i a, ___m256i b)	Unpack and interleave 8-bit integers from the low half of eac
___m256i _mm256_unpacklo_epi16 (___m256i a, ___m256i b)	Unpack and interleave 16-bit integers from the low half of ea
___m256i _mm256_unpacklo_epi32 (___m256i a, ___m256i b)	Unpack and interleave 32-bit integers from the low half of ea
___m256i _mm256_unpacklo_epi64 (___m256i a, ___m256i b)	Unpack and interleave 64-bit integers from the low half of ea

FIGURE 17 – Byte interleaving with AVX2 and 256bits vectors