x86 Intrinsics Cheat Sheet

Jan Finis
finis@in.tum.de

Bit Operations

Boolean Logic

Bool XOR — SSE, SSE2: `xor` — `__si128,__ps[SSE],__pd` — `__mi xor_si128(__mi a,__mi b)`

Bool AND — SSE, SSE2: `and` — `__si128,__ps[SSE],__pd` — `__mi and_si128(__mi a,__mi b)`

Bool NOT AND — SSE, SSE2: `andnot` — `__si128,__ps[SSE],__pd` — `__mi andnot_si128(__mi a,__mi b)` — `!a & b`

Bool OR — SSE, SSE2: `or` — `__si128,__ps[SSE],__pd` — `__mi or_si128(__mi a,__mi b)`

Bit Shifting & Rotation

Arithmetic Shift Right — SSE2: `sra[i]` — `__epi16-64`
NOTE: Shifts elements right while shifting in sign bits. The `i` version takes an immediate as count. The version without `i` takes the lower 64bit of an SSE register.
`__mi srai_epi32(__mi a,__i32 i)`
`__mi sra_epi32(__mi a,__mi b)`

Logic Shift Left/Right — SSE2: `sl/rl[i]` — `__epi16-64`
NOTE: Shifts elements left/right while shifting in zeros. The `i` version takes an immediate as count. The version without `i` takes the lower 64bit of an SSE register.
`__mi slli_epi32(__mi a,__i i)`
`__mi sll_epi32(__mi a,__mi b)`

Rotate Left/Right — S64; x86: `_[i]rot[w]l/r` — `__(u16-64)`
NOTE: Rotates bits in a *left/right* by a number of bits specified in `i`. The `l` version is for 64-bit ints and the `w` version for 16-bit ints.
`__u64 _lrotl(__u64 a)`

Variable Arithmetic Shift — AVX2: `sl/rav` — `__epi32-64`
NOTE: Shifts elements left/right while shifting in sign bits. Each element in `a` is shifted by an amount specified by the corresponding element in `b`.
`__mi slav_epi32(__mi a,__mi b)`

Variable Logic Shift — AVX2: `sl/rlv` — `__epi32-64`
NOTE: Shifts elements left/right while shifting in zeros. Each element in `a` is shifted by an amount specified by the corresponding element in `b`.
`__mi sllv_epi32(__mi a,__mi b)`

Bit Scan Forward/Reverse — S64; x86: `_bit_scan_forward/reverse` — `__(i32)`
NOTE: Returns the index of the *lowest/highest* bit set in the 32bit int `a`. Undefined if `a` is 0.
`__i32 _bit_scan_forward(__i32 a)`

Selective Bit Moving

Change the position of selected bits, zeroing out remaining ones

Bit Scatter (Deposit) — S64; BMI2: `_pdep` — `__u32-64`
NOTE: Deposit contiguous low bits from `a` to `dst` at the corresponding bit locations specified by mask `m`; all other bits in `dst` are set to zero.
`__u64 _pdep_u64(__u64 a,__u64 m)`

Bit Gather (Extract) — S64; BMI2: `_pext` — `__u32-64`
NOTE: Extract bits from `a` at the corresponding bit locations specified by mask `m` to contiguous low bits in `dst`; the remaining upper bits in `dst` are set to zero.
`__u64 _pext_u64(__u64 a,__u64 m)`

Movemask — SSE2: `movemask` — `__epi8,__pd,__ps[SSE]`
NOTE: Creates a bitmask from the most significant bit of each element.
`__i movemask_epi8(__mi a)`

Extract Bits — S64; BMI1: `bextr` — `__u32-64`
NOTE: Extracts `i` bits from `a` starting at bit `s`.
`__u64 _bextr_u64(__u64 a,__u32 s,__u32 i)` — `(a >> s) & ((1 << i)-1)`

Bit Masking

Set or reset a range of bits

Zero High Bits — S64; BMI2: `_bzhi` — `__u32-64`
NOTE: Zeros all bits in `a` higher than and including the bit at index `i`.
`__u64 _bzhi_u64(__u64 a,__u32 i)`
`dst := a`
`IF i[7:0] < 64`
`dst[63:n] := 0`

Reset Lowest 1-Bit — S64; BMI1: `blsr` — `__u32-64`
NOTE: Returns `a` but with the lowest 1-bit set to 0.
`__u64 _blsr_u64(__u64 a)` — `(a-1) & a`

Mask Up To Lowest 1-Bit — S64; BMI1: `blsmsk` — `__u32-64`
NOTE: Returns an int that has all bits set up to and including the lowest 1-bit in `a` or no bit set if `a` is 0.
`__u64 _blsmsk_u64(__u64 a)` — `(a-1) ^ a`

Find Lowest 1-Bit — S64; BMI1: `blsi` — `__u32-64`
NOTE: Returns an int that has only the lowest 1-bit in `a` or no bit set if `a` is 0.
`__u64 _blsi_u64(__u64 a)` — `(-a) & a`

Bit Counting

Count specific ranges of 0 or 1 bits

Count 1-Bits (Popcount) — S64; POPCNT: `popcnt` — `__u32-64`
NOTE: Counts the number of 1-bits in `a`.
`__u32 _popcnt_u64(__u64 a)`

Count Leading Zeros — S64; LZCNT: `_lzcnt` — `__u32-64`
NOTE: Counts the number of leading zeros in `a`.
`__u64 _lzcnt_u64(__u64 a)`

Count Trailing Zeros — S64; BMI1: `_tzcnt` — `__u32-64`
NOTE: Counts the number of trailing zeros in `a`.
`__u64 _tzcnt_u64(__u64 a)`

Conversions

Packed Conversions

Convert all elements in a packed SSE register

Convert Float 16bit ↔ 32bit — CVT16: `cvtX_Y` — `__ph ↔ __ps`
NOTE: Converts between 4x 16 bit floats and 4x 32 bit floats (or 8 for 256 bit mode). For the 32 to 16-bit conversion, a rounding mode `r` must be specified.
`__m cvtph_ps(__mi a)`
`__mi cvtps_ph(__m a, __i r)`

Pack With Saturation — SSE2: `pack[u]s` — `__epi16,__epi32`
NOTE: Packs ints from two input registers into one register halving the bitwidth. Overflows are handled using saturation. The `u` version saturates to the unsigned integer type.
`__mi pack_epi32(__mi a,__mi b)`

Sign Extend — SSE4.1: `cvtX_Y` — `__epi8-32`
NOTE: Sign extends each element from *X* to *Y*. *Y* must be longer than *X*.
`__mi cvtepi8_epi32(__mi a)`

Zero Extend — SSE4.1: `cvtX_Y` — `__epu8-32 → __epi8-32`
NOTE: Zero extends each element from *X* to *Y*. *Y* must be longer than *X*.
`__mi cvtepu8_epi32(__mi a)`

S/D/I32 Conversion — SSE2: `cvt[t]X_Y` — `__epi32,__ps/__pd`
NOTE: Converts packed elements from *X* to *Y*. If `pd` is used as source or target type, then 2 elements are converted, otherwise 4. The `t` version is only available when casting to int and performs a truncation instead of rounding.
`__mi cvtpi32_pd(__mi a)`

Reinterpet Casts

128bit Cast — 128; SSE2: `castX_Y` — `__si128,__ps/__pd`
NOTE: Reinterpret casts from *X* to *Y*. No operation is generated.
`__md castsi128_pd(__mi a)`

128/256bit Cast — 256; AVX: `castX_Y` — `__pd128→__pd256, __ps128→__ps256, __si128→__si256`
NOTE: Reinterpret casts from *X* to *Y*. If cast is from 128 to 256 bit, then the upper bits are undefined! No operation is generated.
`__m128d castpd256_pd128(__m256d a)`

256bit Cast — 256; AVX: `castX_Y` — `__ps,__pd,__si256`
NOTE: Reinterpret casts from *X* to *Y*. No operation is generated.
`__m256 castpd_ps(__m256d a)`

Rounding

See also: Conversion to int performs rounding implicitly

Round up (ceiling) — SSE4.1: `ceil` — `__ps/__pd,__ss/__sd`
`__md ceil_pd(__md a)`

Round down (floor) — SSE4.1: `floor` — `__ps/__pd,__ss/__sd`
`__md floor_pd(__md a)`

Round — SSE4.1: `round` — `__ps/__pd,__ss/__sd`
NOTE: Rounds according to a rounding parameter `r`.
`__md round_pd(__md a,__i r)`

Single Element Conversion

Convert a single element in the lower bytes of an SSE register

Single Conversion to Float with Fill — 128; SSE2: `cvtX_Y` — `__si32-64,__ss/__sd → __ss/__sd`
NOTE: Converts a single element in `b` from *X* to *Y*. The remaining bits of the result are copied from `a`.
`__md cvtsi64_sd(__md a,__i64 b)` — `__double[0:63]: b[63:0]; [127:64]: a[127:64]`

Single Float to Int Conversion — 128; SSE: `cvt[t]X_Y` — `__ss/d → __si32-64`
NOTE: Converts a single element from *X* (int) to *Y* (float). Result is normal int, not an SSE register! The `t` version performs a truncation instead of rounding.
`__i cvtss_si32(__m a)`

Single 128-bit Int Conversion — 128; SSE2: `cvtX_Y` — `__si32-128`
NOTE: Converts a single integer from *X* to *Y*. Either of the types must be `__si128`. If the new type is longer, the integer is zero extended.
`__mi cvtsi32_si128(__i a)`

Single SSE Float to Normal Float Conversion — 128; SSE2: `cvtX_Y` — `__ss→f32,__sd→f64`
NOTE: Converts a single SSE float from *X* (SSE Float type) to *Y* (normal float type).
`__f cvtss_f32(__m a)`

Old Float/Int Conversion — 128; SSE: `cvt[t]_X2Y` — `__pi→ps,__si→ss`
NOTE: Converts *X* to *Y*. Converts between int and float. `_p_` versions convert 2-packs from `b` and fill result with upper bits of operand `a`. `_s_` versions do the same for one element in `b`. The `t` version truncates instead of rounding and is available only when converting from float to int.
`__m cvt_si2ss(__m a,__i b)`

Register I/O

Loading data into an SSE register or storing data from an SSE register

Load

Loading data into an SSE register

Set Register

Inserting data into an register without loading from memory

Set Reversed — SEQ?, SSE2, AVX: `setr` — `__epi8-64x,__ps[SSE],__pd, __m128/__m128d/__i[AVX]`
NOTE: Sets and returns an SSE register with input values. The order in the register is reversed, i.e. the first input gets stored in the lowest bits. For `epi64`, use the `epi64x` suffix.
`__mi setr_epi32(__i a,__i b,__i c,__i d)`

Set — SEQ?, SSE2, AVX: `set` — `__epi8-64x, __ps[SSE]/__d,__ss/d, __m128/__m128d/__i[AVX]`
NOTE: Sets and returns an SSE register with input values. E.g., `epi8` version takes 16 input parameters. The first input gets stored in the highest bits. For `epi64`, use the `epi64x` suffix.
`__mi set_epi32(__i a,__i b,__i c,__i d)`

Insert — 128; SSE4.1: `insert` — `__epi16[SSE2], __epi8-64,__ps`
NOTE: Inserts an element `i` at a position `p` into `a`.
`__mi insert_epi16(__mi a,__i i,__i p)`
`dst[127:0] := a[127:0]`
`sel := p[2:0]*16`
`dst[sel+15:sel]:=i[15:0]`

Replicate — SEQ?; SSE2: `set1` — `__epi8-64x,__ps[SSE]/__d`
NOTE: Broadcasts one input element into all slots of an SSE register.
`__mi set1_epi32(__i a)`

Aligned Load

Loading data from a memory address which must be 16-byte aligned (or 32-byte for 256bit instructions)

Stream Load — SSE2: `stream_load` — `__si128,__si256[AVX]`
NOTE: Loads 128 bit from memory into register. Memory address must be aligned! Memory is fetched at once from an USWC device without going through the cache hierachy.
`__mi __stream_load_si128(__mi* ptr)`

Load Aligned — SSE2: `load` — `__pd,__ps,__si128`
NOTE: Loads 128 bit from memory into register. Memory address must be aligned!
`__md __load_pd(__d* ptr)`

Load Reversed — 128; SEQ?, SSE2: `loadr` — `__pd,__ps`
NOTE: Loads 128 bit from memory into register. The elements are loaded in reversed order. Memory must be aligned!
`__m loadr_ps(__f* ptr)`
`dst[31:0]:=*(ptr)[127:96]`
`dst[63:32]:=*(ptr)[95:64]`
`dst[95:64]:=*(ptr)[63:32]`
`dst[127:96]:=*(ptr)[31:0]`

Mask Load — AVX, AVX2: `maskload` — `__ps/__d,__epi32-64[AVX2]`
NOTE: Loads from memory if the highest bit for each element is set in the mask. Otherwise 0 is used. `ptr` must be aligned!
`__mi maskload_epi64(__i64* ptr,__mi mask)`
`dst[MAX:128] := 0`
`FOR j := 0 to 1`
`i := j*64`
`IF mask[i+63]`
`dst[i+63:i]:= *(ptr+i)`
`ELSE`
`dst[i+63:i]:= 0`

Unaligned Load

Loading data from a memory address which does not have to be aligned to a specific boundary

Fast Load Unaligned — SSE3: `lddqu` — `__si128`
NOTE: Loads 128bit integer data into SSE register. Is faster than `loadu` if value crosses cache line boundary. Should be preferred over `loadu`.
`__mi __lddqu_si128(__mi* ptr)`

Load High/Low — 128; SSE2: `loadh/l` — `__pd,__ps`
NOTE: Loads a value from memory into the *high/low* half of a register and fills the other half from `a`.
`__md __loadl_pd(__md a,__d* ptr)`
`dst[63:0] := *(ptr)[63:0]`
`dst[127:64] := a[127:64]`

Broadcast Load — 128; SEQ?, SSE2: `load1` — `__pd,__ps`
NOTE: Loads a float from memory into all slots of the 128-bit register. For `pd`, there is also the operation `loaddup` in SSE3 which may perform faster than `load1`.
`__md __load1_pd(__d* ptr)`

Broadcast Load — AVX: `broadcast` — `__ss/d,__ps/d`
NOTE: Broadcasts one input (**ss/d**) element or 128bits (**ps/d**) from memory into all slots of an SSE register. All suffixes but **ss** are only available in 256 bit mode!
`__m broadcast_ss(__f* a)`

Load Unaligned — SSE2: `loadu` — `__pd,__ps,__si16-__si128`
NOTE: Loads 128 bit or less (for si16-64 versions) from memory into the low-bits of a register.
`__md __loadu_pd(__d* ptr)`

Load Single — 128; SSE2: `load` — `__ss,__sd,__epi64`
NOTE: Loads a single element from memory and zeros remaining bytes. For `epi64`, the command is `__loadl_epi64`!
`__md __load_sd(__d* ptr)`

Broadcast Load — 128; SSE3: `loaddup` — `__pd`
NOTE: Loads a 64-bit float from memory into all slots of the 128-bit register.
`__md __loaddup_pd(__d* ptr)`

128bit Pseudo Gather — 256; SEQ?, AVX: `loadu2` — `__m128,__m128d,__m128i`
NOTE: Loads two 128bit elements from two memory locations.
`__m loadu2_m128(__f* p1,__f* p2)`
`dst[127:0] := *p2;`
`dst[255:128] := *p1;`

Fences

Store Fence — SSE: `sfence` — `__-`
NOTE: Guarantees that every store instruction that precedes, in program order, is globally visible before any store instruction which follows the fence in program order.
`__v sfence()`

Load Fence — SSE2: `lfence` — `__-`
NOTE: Guarantees that every load instruction that precedes, in program order, is globally visible before any load instruction which follows the fence in program order.
`__v lfence()`

Memory Fence (Load & Store) — SSE2: `mfence` — `__-`
NOTE: Guarantees that every memory access that precedes, in program order, the memory fence instruction is globally visible before any memory instruction which follows the fence in program order.
`__v mfence()`

Misc I/O

Prefetch — SSE: `prefetch` — `__-`
NOTE: Fetch the line of data from memory that contains address `ptr` to a location in the cache heirarchy specified by the locality hint `i`.
`__v prefetch(__c* ptr,__i i)`

Cache Line Flush — SSE2: `clflush` — `__-`
NOTE: Flushes cache line that contains `p` from all cache levels
`__v clflush(__v* ptr)`

Get Undefined Register — AVX: `undefined` — `__ps/__d,__si128-256`
NOTE: Returns an SSE register with undefined contents.
`__mi __undefined_si128()`

Store

Storing data from an SSE register into memory or registers

Aligned Store

Storing data to a memory address which must be 16-byte aligned (or 32-byte for 256bit instructions)

Aligned Stream Store — SSE, SSE2, AVX: `stream` — `__si32-128,__pd,__ps[SSE], __si256[AVX]`
NOTE: Stores 128-bits (or 32-64) for `__si32-64` integer a into memory using a non-temporal hint to minimize cache pollution. Memory location must be aligned!
`__v __stream_si32(__i* p,__i a)`
`__v __stream_si128(__m* p,__mi a)`

Aligned Reverse Store — 128; SEQ?, SSE, SSE2: `storer` — `__pd,__ps[SSE]`
NOTE: Stores the elements from the register to memory in reverse order. Memory must be aligned!
`__v __storer_pd(__d* ptr,__md a)`

Aligned Store — SSE, SSE2: `store` — `__si128,__pd,__ps[SSE], __si256[AVX]`
NOTE: Stores 128-bits into memory. Memory location must be aligned!
`__v __store_pd(__d* ptr,__md a)`

Broadcast Store — 128; SEQ?, SSE, SSE2: `store1` — `__pd,__ps[SSE]`
NOTE: Stores a float to 128-bits of memory replicating it two (`pd`) or four (`ps`) times. Memory location must be aligned!
`__v __store1_pd(__d* ptr,__md a)`

Masked Store — AVX, AVX2: `maskstore` — `__ps/__d,__epi32-64[AVX2]`
NOTE: Stores bytes from `a` into memory at `p` if the corresponding byte in the mask `n` has its highest bit set. Memory must be aligned!
`__v __maskstore_ps(__f* p,__mi m,__m a)`

Unaligned Store

Storing data to a memory address which does not have to be aligned to a specific boundary

Unaligned Store — SSE, SSE2: `storeu` — `__si16-128,__pd,__ps[SSE], __si256[AVX]`
NOTE: Stores 128 bits (or less for the si16-64 versions) into memory.
`__v __storeu_pd(__v* ptr,__m a)`

Store High — 128; SSE2: `storeh` — `__pd`
NOTE: Stores the high 64bits into memory.
`__v __storeh_pd(__v* ptr,__m a)`

Single Element Store — 128; SSE, SSE2: `store[l]` — `__ss[SSE],__sd,__epi64`
NOTE: Stores the low element into memory. The `l` version must be used for `epi64` and can be used for `pd`.
`__v __store_sd(__d* ptr,__md a)`

Masked Store — SSE2: `maskmoveu` — `__si128`
NOTE: Stores bytes into memory if the corresponding byte in `mask` has its highest bit set.
`__v __maskmoveu_si128(__mi a,__mi mask,__c* ptr)`

128bit Pseudo Scatter — 256; SEQ?, AVX: `storeu2` — `__m128,__m128d,__m128i`
NOTE: Stores two 128bit elements into two memory locations.
`__v __storeu2_m128(__f* p,__f* q,__m a)`
`*p = dst[127:0];`
`*q = dst[255:128];`

Extraction

Extract — 128; SSE4.1: `extract` — `__epi16[SSE2], __epi8-64,__ps`
NOTE: Extracts one element and returns it as a normal int (no SSE register!)
`__i __extract_epi16(__mi a,__ii i)`
`(a>>(i[2:0]*16))[15:0]`

256bit Extract — AVX, AVX2: `extractf128` — `__si256,__ps/__d`
NOTE: Extracts 128bit and returns it as a 128bit SSE register. For AVX2, there is only `__si256` and the operation is named `__extracti128`.
`__m128 __extractf128_ps(__m256 a,__ii i)`
`(a >> (i * 128))[128:0]`

Overview

Version 2.1f

Introduction

This cheat sheet displays most x86 intrinsics supported by Intel processors. The following intrinsics were omitted:

- obsolete or discontinued instruction sets like MMX and 3DNow!
- AVX-512, as it will not be available for some time and would blow up the space needed due to the vast amount of new instructions
- Intrinsics not supported by Intel but only AMD like parts of the XOP instruction set (maybe they will be added in the future).
- Intrinsics that are only useful for operating systems like the `_xsave` intrinsic to save the CPU state
- The RdRand intrinsics, as it is unclear whether they provide real random numbers without enabling kleptographic loopholes

Each family of intrinsics is depicted by a box as described below. It was tried to group the intrinsics meaningfully. Most information is taken from the Intel Intrinsics Guide (http://software.intel.com/en-us/articles/intel-intrinsics-guide). Let me know (finis@in.tum.de) if you find any wrong or unclear content.

When not stated otherwise, it can be assumed that each vector intrinsic performs its operation *vertically* on all elements which are packed into the input SSE registers. E.g., the add instruction has no description. Thus, it can be assumed that it performs a vertical add of the elements of the two input registers, i.e., the first element from register **a** is added to the first element in register **b**, the second is added to the second, and so on. In contrast, a *horizontal* add would add the first element of **a** to the second element of **a**, the third to the fourth, and so on.

To use the intrinsics, included the `<x86intrin.h>` header and make sure that you set the target architecture to one that supports the intrinsics you want to use (using the `-march=X` compiler flag).

Name: Human readable name of the operation

Sequence Indicator: If this indicator is present, it means that the intrinsic will generate a sequence of assembly instructions instead of only one. Thus, it may not be as efficient as anticipated.

Instruction Set: Specifies the instruction set which is necessary for this operation. If more than one instruction set is given here, then different flavors of this operation require different instruction sets.

Notes: Explains the semantics of the intrinsic, the various flavors, and other important information.

Float Compare — SEQ?, SSE2: `cmp[n]Z` — `__ps/__d,__ss/__d, __si128[SSE4.1]`
NOTE: *Z* can be one of: `ge/le/lt/gt/eq`. Returns a single int that is either 1 or 0. The `n` version is a not version, e.g., `neq` computes not equal.
`__md cmpeq_pd(__md a,__md b)`

Available Bitwidth: If no bitwidth is specified, the operation is available for 128bit and 256bit SSE registers. Use the `__mm_` prefix for the 128bit and the `__mm256_` prefix for the 256bit flavors. Otherwise, the following restrictions apply:

256 : Only available for 256bit SSE registers (always use `__mm256_` prefix)
128 : Only available for 128bit SSE registers (always use `__mm_` prefix)
S64 : Operation does not operate on SSE registers but usual 64bit registers (always use `__mm_` prefix)

Name(s) of the intrinsic: The names of the various flavors of the intrinsic. To assemble the final name to be used for an intrinsic, one must add a prefix and a suffix. The suffix determines the data type (see next field). Concerning the prefix, all intrinsics, except the ones which are prefixed with a green underscore `_`, need to be prefixed with `__mm_` for 128bit versions or `__mm256_` for 256bit versions. Blue letters in brackets like `[n]` indicate that adding this letter leads to another flavor of the function. Blue letters separated with a slash like `l/r` indicate that either letter can be used and leads to a different flavor. The different flavors are explained in the notes section. A red letter like *Z* indicates that various different strings can be inserted here which are stated in the notes section.

List of available data type suffixes: Consult the suffix table for further information about the various possible suffixes. The suffix chosen determines the data type on which the intrinsic operates. It must be added as a suffix to the intrinsic name separated by an underscore, so a possible name for the data type `pd` in this example would be `__mm_cmpeq_pd`. A suffix is followed by an instruction set in brackets, then this instruction set is required for the suffix. All suffixes without an explicit instruction set are available in the instruction set specified at the left. If no type suffixes are shown, then the method is type independent and must be used without a suffix. If the suffixes are in parenthesis, the suffixes must not be appended and are encoded into the name of the intrinsic in another way (see notes for further information).

Signature and Pseudocode: Shows one possible signature of the intrinsic in order to depict the parameter types and the return type. Note that only one flavor and data type suffix is shown; the signature has to be adjusted for other suffixes and falvors. The data types are displayed in shortened form. Consult the data types table for more information.

In addition to the signature, the pseudocode for some intrinsics is shown here. Note that the special variable `dst` depicts the destination register of the method which will be returned by the intrinsic.

Instruction Sets

Each intrinsic is only available on machines which support the corresponding instruction set. This list depicts the instruction sets and the first Intel and AMD CPUs that supported them.

I.Set	Intel *(Year)*	AMD *(Year)*	Description
x86	all	all	x86 Base Instructions
SSE	Pentium III *99*	K7 Palomino *01*	Streaming SIMD Extensions
SSE2	Pentium 4 *01*	K8 *03*	
SSE3	Prescott *04*	K9 *05*	
SSSE3	Woodcrest *06*	Bulldozer *11*	
SSE4.1	Penryn *07*	Bulldozer *11*	
SSE4.2	Nehalem *08*	Bulldozer *11*	
AVX	Sandy Bridge *11*	Bulldozer *11*	Advanced Vector Extensions
AVX2	Haswell *13*	-	
FMA	Haswell *13*	Bulldozer *11*	Fused Multiply add
AES	Westmere *10*	Bulldozer *11*	Advanced Encryption Standard
CLMUL	Westmere *10*	Bulldozer *11*	Carryless Multiplication
CVT16	Ivy Bridge *12*	Bulldozer *11*	16-bit Floats
TSX	Haswell *13*	-	Transactional Sync. Extensions
POPCNT	Nehalem *08*	K10 *07*	Bit Manipulation Instructions
LZCNT	Haswell *13*	K10 *07*	
BMI1	Sandy Bridge *11*	Bulldozer *11*	
BMI2	Haswell?? *14*	-	

Data Type Suffixes

Most intrinsics are available for various suffixes which depict different data types. The table depicts the suffixes used and the corresponding SSE register types (see Data Types for descriptions).

Suffix	Type	Description
ph	mi	Packed half float (16 bit float)
ps/d	m/md	Packed single float / packed double float
epiX	mi	Packed X bit signed integer
epuX	mi	Packed X bit unsigned integer
siX	mi	Single X bit signed integer. If there is a si128 version, the same function for 256 bits is usually suffixed with si256.
suX	mi	Single X bit unsigned integer in SSE register
uX	-	Single X bit unsigned integer (not in SSE register)
ss/d	m/md	Single single float/double float, remaining bits copied. Operations involving these types often have different signatures: An extra input register is used and all bits which are not used by the single element are copied from this register. The signatures are not depicted here; see manual for exact signatures.
m128, m128d, m128i	m, md, mi	AVX sometimes uses this suffix for 128bit float/double/int operations.

Data Types

The following data types are used in the signatures of the intrinsics. Note that most types depend on the used type suffix and only one example suffix is shown in the signature.

Suffix	Description
i	Signed 32 bit integer (`int`)
iX	Signed *X* bit integer (`intX_t`)
uX	Unsigned *X* bit integer (`uintX_t`)
ii	Immediate signed 32 bit integer: The value used for parameters of this type must be a compile time constant
f	32-bit float (`float`)
d	64-bit float (`double`)
mi	Integer SSE register, i.e., `__m128i` or `__m256i`, depending on the bitwidth used.
m	32-bit Float SSE register, i.e., `__m128` or `__m256`, depending on the bitwidth used.
md	64-bit Float (double) SSE register, i.e., `__m128d` or `__m256d`, depending on the bitwidth used.
v	`void`
X*	Pointer to *X* (*X)

Byte Manipulation

Change the order of bytes in a register, duplicating or zeroing bytes selectively or mixing the bytes of two registers

Mix Registers

Mix the contents of two registers

Move Element with Fill — 128; SSE, SSE2: `move` — `__ss[SSE],__sd`
NOTE: Moves the lowest element from first input and fills remaining elements from second input
`__m move_ss(__m a,__m b)`
`a[31:0] | b[127:32]`

Move High↔Low — 128; SSE: `movelh/hl` — `__ps`
NOTE: The `lh` version moves lower half of `b` into upper half of result. Rest is filled from `a`. The `hl` version moves upper half to lower half.
`__m movehl_ps(__m a,__m b)`

256bit Insert — 256; AVX, AVX2: `insertf128` — `__si256,__ps/__d`
NOTE: Inserts 128bit at a position specified by `i`. For AVX2, there is only `__si256` and the operation is named `__inserti128`.
`__m insertf128_ps(__m a,__m b,__i i)`
`dst[255:0] := a[255:0]`
`sel := i*128`
`dst[sel+15:sel]:=b[127:0]`

Concatenate and Byteshift (Align) — SSSE3: `alignr` — `__epi8`
`__mi alignr_epi8(__mi a,__mi b,__i c)`
`((a << 128) | b) >> c*8`

128-bit Dual Register Shuffle — 256; AVX, AVX2: `permute2f128` — `__ps/__d,__si256`
NOTE: Takes 2 registers and shuffles 128 bit chunks to a target register. Each chunk can also be cleared by a bit in the mask. For AVX2, there is only the `__si256` version which is renamed to `__m256_permute2x128_pd`.
`__m256 __m256_permute2f128_pd(__m a,__m b,__i i)`
`S(s1, s2, control){`
`CASE(control[1:0])`
`0: tmp[127:0]:=s1[127:0]`
`1: tmp[127:0]:=s1[255:128]`
`2: tmp[127:0]:=s2[127:0]`
`3: tmp[127:0]:=s2[255:128]`
`IF control[3]`
`tmp[127:0] := 0`
`RETURN tmp[127:0]`
`}`
`dst[127:0] := S(a,b,i[3:0])`
`dst[255:128]:=S(a,b,i[7:4])`
`dst[MAX:256]:=0`

Interleave (Unpack) — SSE2: `unpack[h/l]` — `__epi8-64,__ps[SSE],__pd`
NOTE: Interleaves elements from the *high/low* half of two input registers into a target register.
`__mi unpackhi_epi32(__mi a,__mi b)`
`dst[31:0] := a[95:64]`
`dst[63:32] := b[95:64]`
`dst[95:64] := a[127:96]`
`dst[127:96] := b[127:96]`

Blend — SSE4.1, AVX2: `blend[v]` — `__epi8-16, __epi32[AVX2],__ps/__d`
NOTE: `blendv` uses 128bit mask, `blend` uses immediate
`__mi blend_epi16(__mi a,__mi b,__ii imm)`
`FOR j := 0 to 7`
`i := j*16`
`IF imm[j]`
`dst[i+15:i] := b[i+15:i]`
`ELSE`
`dst[i+15:i] := a[i+15:i]`

Dual Register Float Shuffle — SSE, SSE2: `shuffle` — `__ps[SSE],__pd`
NOTE: Shuffles floats from two registers. The lower part of the result receives values from the first register. The upper part receives values from the second register. Shuffle mask is an *immediate*!
`__mi shuffle_pd(__md a,__md b,__ii i)`
`dst[63:0]:= (i[0] == 0) ? a[63:0] : a[127:64]`
`dst[127:64]:= (i[1] == 0) ? b[63:0] : b[127:64]`

Byte Shuffling

Change the byte order using a control mask

32-bit Int Shuffle — SSE2: `shuffle` — `__epi32`
`__mi shuffle_epi32(__mi a,__ii i)`
`S(a, mask){`
`CASE(mask[1:0])`
`0: r := s[31:0]`
`1: r := s[63:32]`
`2: r := s[95:64]`
`3: r := s[127:96]`
`RETURN r[31:0]`
`}`
`dst[31:0]:=S(a,i[1:0])`
`dst[63:32]:=S(a,i[3:2])`
`dst[95:64]:=S(a,i[5:4])`
`dst[127:96]:=S(a,i[7:6])`

High / Low 16bit Shuffle — SSE2: `shuffle[h/l]o` — `__epi16`
NOTE: Shuffles the *high/low* half of the register using an immediate control mask. Rest of the register is copied from input.
`__mi shufflehi_epi16(__mi a,__ii i)`
`dst[63:0] := a[63:0]`
`dst[79:64] :=`
`(a >> ((i[3:0]*16))[79:64]`
`dst[95:80] :=`
`(a >> ((i[3:2]*16))[79:64]`
`dst[111:96] :=`
`(a >> ((i[5:4]*16))[79:64]`
`dst[127:112] :=`
`(a >> ((i[7:6]*16))[79:64]`

Byte Shuffle — SSSE3: `shuffle` — `__epi8`
NOTE: Shuffle packed 8-bit integers in a according to shuffle control mask in the corresponding 8-bit element of `b`, and store the results in `dst`.
`__mi shuffle_epi8(__mi a,__mi b)`
`FOR j := 0 to 15`
`i := j*8`
`IF b[i+7] == 1`
`dst[i+7:i] := 0`
`ELSE`
`k := b[i+3:i]*8`
`dst[i+7:i]:= a[k+7:k]`

Float Shuffle — AVX: `permute[var]` — `__ps/__d`
NOTE: 128-bit version is the same as int shuffle for floats. 256-bit version performs the same operation on two 128-bit lanes. The normal version uses an immediate and the `var` version a register `b` (the lowest bits in each element in `b` are used for the mask of that element in `a`).
`__m permutevar_ps(__m a,__mi b)`

4x64bit Shuffle — 256; AVX2: `permute4x64` — `__epi64,__pd`
NOTE: Same as float shuffle but no `[var]` version is available. In addition, the shuffle is performed over the full 256 bit instead of two lanes of 128 bit.
`__md permute4x64_pd(__md a,__ii i)`

8x32bit Shuffle — 256; AVX2: `permutevar8x32` — `__epi32,__ps`
NOTE: Same as 4x64 bit shuffle but with 8x32bit and only a `[var]` version taking a register instead of an immediate is available.
`__m permutevar8x32_ps(__m a,__mi b)`

Byte Zeroing

Zero Register — SSE, SSE2: `setzero` — `__ps[SSE],__d,__si128`
NOTE: Returns a register with all bits zeroed.
`__mi setzero_si128()`

Zero All Registers — 256; AVX: `zeroall` — `__-`
NOTE: Zeros all SSE/AVX registers.
`__v zeroall()`

Zero High — 128; SSE2: `move` — `__epi64`
NOTE: Moves lower half of input into result and zero remaining bytes.
`__mi move_epi64(__mi a)`
`a[63:0]`

Zero High All Registers — 256; AVX: `zeroupper` — `__-`
NOTE: Zeros all bits in [MAX:256] of all SSE/AVX registers.
`__v zeroupper()`

Broadcast

Replicating one element in a register to fill the entire register

64-bit Broadcast — SSE3: `movedup` — `__pd`
NOTE: Duplicates the lower half of `a`.
`__md __movedup_pd(__md a)`
`a[0:63]|a[63:0]<<64)`

32-bit Broadcast High/Low — SSE3: `movel/hdup` — `__ps`
NOTE: Duplicates 32bits into the lower 64bits of the result. `l` version duplicates bits [31:0], `h` version duplicates bits [63,32].
`__m __moveldup_ps(__m a)`

Broadcast — AVX2: `broadcastX` — `__epi8-64,__ps/__d,__si256`
NOTE: Broadcasts the lowest element into all slots of the result register. The `si256` version broadcasts one 128bit element! The letter *X* must be the following: `epi8`: `b`, `epi16`: `w`, `epi32`: `d`, `epi64`:`q`, `ps`: `s`, `pd`: `d`, `si256`: `si128`
`__mi __broadcastb_epi8(__mi a)`

Byte Movement

Byteshift left/right — SSE2: `[b]sl/rli` — `__si128,__epi128[256]`
NOTE: Shift input register left/right by `i` bytes while shifting in zeros. There is no difference between the `b` and the non-`b` version.
`__mi __bslli_si128(__mi a,__ii i)`

Byte Swap — S64; x86: `bswap[64]` — `__(i32,i64)`
NOTE: Swaps the bytes in the 32-bit int or 64-bit int.
`__i64 __bswap64(__i64 a)`

Arithmetics

Basic Arithmetics

Multiplication

Carryless Mul — 128; CLMUL: `clmulepi64` — `__si128`
NOTE: Perform a carry-less multiplication of two 64-bit integers, selected from **a** and **b** according to **imm**, and store the results in dst (result is a 127 bit int).
`__mi clmulepi64_si128(__mi a,__mi b,__ii imm)`

Mul — SSE, SSE2: `mul` — `__epi32[SSE4.1],__epu32, __ps/__d,__ss/__d`
NOTE: `epi32` and `epu32` version multiplies only 2 ints instead of 4!
`__mi mul_epi32(__mi a,__mi b)`

Mul Low — SSE2, SSE4.1: `mullo` — `__epi16,__epi32[SSE4.1]`
NOTE: Multiplies vertically and writes the low 16/32 bits into the result.
`__mi mullo_epi16(__mi a,__mi b)`

Mul High — SSE2: `mulhi` — `__epi16,__epu16`
NOTE: Multiplies vertically and writes the high 16bits into the result.
`__mi mulhi_epi16(__mi a,__mi b)`

Mul High with Round & Scale — SSSE3: `mulhrs` — `__epi16`
NOTE: Treat the 16-bit words in registers **a** and **b** as signed 15-bit fixed-point numbers between −1 and 1 (e.g. 0x4000 is treated as 0.5 and 0xa000 as −0.75), and multiply them together.
`__mi mulhrs_epi16(__mi a,__mi b)`

Addition / Subtraction

Horizontal Add with Saturation — SSSE3: `hadds` — `__epi16`
NOTE: Adds adjacent pairs of elements with saturation
`__mi hadds_epi16(__mi a,__mi b)`

Horizontal Add — SSSE3: `hadd` — `__epi16-32,__ps/__d`
NOTE: Adds adjacent pairs of elements
`__mi hadd_epi16(__mi a,__mi b)`

**