x86 Intrinsics Cheat Sheet

Jan Finis
finis@in.tum.de

Legend

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.

Float Compare

SEQ! | 128
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)
- ≤64: 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.

Notes: Explains the semantics of the intrinsic, the various flavors, and other important 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. The green numbers behind the CPUs depict the year of release.

I.Set	Intel	AMD	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 and 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 Table for descriptions).

Suffix	Reg.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 for 128bit single/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*`)

Specials

Special Algorithms

AES KeyGen Assist
128 | AES `aeskeygenassist` | si128
NOTE: Assist in expanding the AES cipher key by computing steps towards generating a round key for encryption cipher using data from **a** and an 8-bit round constant specified in **imm**.
`mi aeskeygenassist_si128(mi a,ii imm)`

AES Inverse Mix Columns
128 | AES `aesimc` | si128
NOTE: Performs the inverse mix columns transformation on the input.
`mi aesimc_si128(mi a)`

AES Encrypt
128 | AES `aesenc[last]` | si128
NOTE: Perform one round of an AES encryption flow on data (state) in a using the round key. The **last** version performs the last round.
`mi aesenc_si128(mi a,mi key)`

AES Decrypt
128 | AES `aesdec[last]` | si128
NOTE: Perform one round of an AES decryption flow on data (state) in a using the round key. The **last** version performs the last round.
`mi aesdec_si128(mi a,mi key)`

Cyclic Redundancy Check (CRC32)
≤64 | SSE4.2 `crc32` | u8-u64
NOTE: Starting with the initial value in crc, accumulates a CRC32 value for unsigned X-bit integer v, and stores the result in dst.
`u crc32_u32(u crc,u v)`

Transactional Memory

Abort Transaction
TSX `_xabort` | -
NOTE: Force an RTM abort. The EAX register is updated to reflect an XABORT instruction caused the abort, and the imm parameter will be provided in bits [31:24] of EAX. Following an RTM abort, the logical processor resumes execution at the fallback address computed through the outermost XBEGIN instruction.
`v _xabort(ii imm)`

Commit Transaction
TSX `_xend` | -
NOTE: Specify the end of an RTM code region. If this corresponds to the outermost scope, the logical processor will attempt to commit the logical processor state atomically. If the commit fails, the logical processor will perform an RTM abort.
`v _xend()`

Begin Transaction
TSX `_xbegin` | -
NOTE: Specify the start of an RTM code region. If the logical processor was not already in transactional execution, then it transitions into transactional execution. On an RTM abort, the logical processor discards all architectural register and memory updates performed during the RTM execution, restores architectural state, and starts execution beginning at the fallback address computed from the outermost XBEGIN instruction.
`u _xbegin()`

Miscellaneous

Pause
SSE2 `pause` | -
NOTE: Provide a hint to the processor that the code sequence is a spin-wait loop. This can help improve the performance and power consumption of spin-wait loops.
`v pause()`

Monitor Memory
SSE3 `monitor` | -
NOTE: Arm address monitoring hardware using the address specified in p. A store to an address within the specified address range triggers the monitoring hardware. Specify optional extensions in ext, and optional hints in hints.
`v monitor(v* ptr,u ext,u hints)`

Monitor Wait
SSE3 `mwait` | -
NOTE: Hint to the processor that it can enter an implementation-dependent-optimized state while waiting for an event or store operation to the address range specified by MONITOR. ??????
`v mwait(u ext,u hints)`

Get MXCSR Register
SSE `getcsr` | -
NOTE: Get the content of the MXCSR control and status register.
`u getcsr()`

Set MXCSR Register
SSE `setcsr` | -
NOTE: Set the MXCSR control and status register with a 32bit int.
`v setcsr(u a)`

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[SSE1]/d, m128/m128d/i[AVX]
NOTE: Sets and returns an SSE register with input values. E.g., epi8 version takes 16 input parameters. The order in the register is reversed, i.e. the first input gets stored in the lowest bits. The 128bit epi64 version takes two MMX registers. For 256bit, there is only an epi64x version which takes usual ints.
`mi setr_epi32(i a,i b,i c,i d)`

Set
SEQ! | SSE2, AVX `set` | epi8-64x, ps[SSE1]/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.
`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 at a specified position.
```
mi insert_epi16(mi a,i i,ii imm)
dst[127:0] := a[127:0]
sel := imm[2:0]*16
dst[sel+15:sel]:=i[15:0]
```

Replicate
SEQ! | SSE2 `set1` | epi8-64x,ps[SSE1]/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
SEQ! | 128 | 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! 128bit versions zero upper parts of the register.
```
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 Unaligned Load
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,pi
NOTE: Loads a value from memory into the high/low half of the register and fills the other half from the input register. Memory does not need to be aligned.
```
md loadl_pd(md a,d* ptr)
dst[63:0] :=*(ptr)[63:0]
dst[127:64] := a[127:64]
```

Broadcast Load
SEQ! | 128 | 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. Memory does not need to be aligned.
`md load1_pd(d* ptr)`

Broadcast Load
128 | SSE3 `loaddup` | pd
NOTE: Loads a 64-bit float from memory into all slots of the 128-bit register. Memory does not need to be aligned.
`md loaddup_pd(d* ptr)`

Gather
AVX2 `i32/i64gather` | epi32-64,ps/d
NOTE: Gather elements from memory using 32-bit/64-bit indices. The elements are loaded from addresses starting at ptr and offset by each 32-bit/64-bit element in a (each index is scaled by the factor in s). Gathered elements are merged into dst. s should be 1, 2, 4 or 8. The 128bit versions zero the upper 128bit of 256bit registers. The number of gathered elements is limited by the minimum of the type to load and the type used for the offset. Memory does not need to be aligned.
```
mi i64gather_epi32
(i* ptr,mi a,i s)
FOR j := 0 to 1;
 i := j*32
 m := j*64
 dst[i+31:i] :=
   *(ptr + a[m+63:m]*s])
dst[MAX:64] := 0
```

Mask Gather
AVX2 `mask_i32/i64gather` | epi32-64,ps/d
NOTE: Same as gather but takes an additional mask and src register. Each element is only gathered if the highest corresponding bit in the mask is set. Otherwise it is copied from src. Memory does not need to be aligned.
```
mi mask_i64gather_epi32 (mi src,
i* ptr,mi a,mi mask,i32 s)
FOR j := 0 to 1; i:=j*32;m:=j*64
 IF mask[i+31]
  dst[i+31:i]:=*(ptr+a[i+63:i]*s)
  mask[i+31] := 0
 ELSE
  dst[i+31:i] := src[i+31:i]
mask[MAX:64] := 0
dst[MAX:64] := 0
```

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. Memory does not have to be aligned.
`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! Memory does not need to be aligned.
`md load_sd(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)`

128bit Pseudo Gather
SEQ! | 256 | AVX `loadu2` | m128,m128d,m128i
NOTE: Loads two 128bit elements from two memory locations which do not need to be aligned.
```
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)`

Cashline 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 `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 16-byte aligned!
```
v stream_si32(i* ptr,i a)
v stream_si128(mi* ptr,mi a)
```

Aligned Reverse Store
SEQ! | 128 | SSE, SSE2 `storer` | pd,ps[SSE]
NOTE: Stores the elements from the register to memory in reverse order. Memory must be 16-byte 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 16-byte aligned!
`v store_pd(d* ptr,md a)`

Broadcast Store
SEQ! | 128 | SSE, SSE2 `store1` | pd[SSE2],ps
NOTE: Stores a float to 128-bits of memory replicating it two (pd) or four (ps) times. Memory location must be 16-byte 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 m 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. Memory does not need to be aligned.
`v storeu_pd(v* ptr,m a)`

Store High
128 | SSE2 `storeh` | pd
NOTE: Stores the high 64bits into memory. Memory does not need to be aligned.
`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. Memory does not need to be aligned. 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. Memory does not need to be aligned.
`v maskmoveu_si128(mi a,mi mask,c* ptr)`

128bit Pseudo Scatter
SEQ! | 256 | AVX `storeu2` | m128,m128d,m128i
NOTE: Stores two 128bit elements into two memory locations which do not need to be aligned.
```
v storeu2_m128(f* p1,f* p2,m a)
*p2 = dst[127:0];
*p1 = 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 imm)
(a>>(imm[2:0]*16))[15:0]
```

256bit Extract
256 | 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]
```

Conversions

Packed Conversions

Convert all elements in a packed SSE register

Convert 16bit Float ↔ 32bit Float
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 must be specified.
```
m cvtph_ps(mi a)
mi cvtps_ph(m a, i rounding)
```

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/d
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.
`md cvtepi32_pd(mi a)`

Reinterpet Casts

128bit Cast
128 | SSE2 `castX_Y` | si128,ps/d
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/d,**ss/d**
`md ceil_pd(md a)`

Round down (floor)
SSE4.1 `floor` | ps/d,**ss/d**
`md floor_pd(md a)`

Round
SSE4.1 `round` | ps/d,**ss/d**
NOTE: Rounds according to a rounding paramater.
`md round_pd(md a,i rounding)`

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/d → ss/d
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(b[63:0]) |
  (a[127:64] << 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)`

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)`

Selective Bit Moving

Change the position of selected bits, zeroing out remaining ones

Bit Scatter (Deposit)
≤64 | BMI2 `_pdep` | u32-64
NOTE: Deposit contiguous low bits from a to dst at the corresponding bit locations specified by mask; all other bits in dst are set to zero.
`u64 _pdep_u64(u64 a, u64 mask)`

Bit Gather (Extract)
≤64 | BMI2 `_pext` | u32-64
NOTE: Extract bits from a at the corresponding bit locations specified by mask to contiguous low bits in dst; the remaining upper bits in dst are set to zero.
`u64 _pext_u64(u64 a, u64 mask)`

Movemask
SSE2 `movemask` | epi8,pd,ps[SSE]
NOTE: Creates an int bitmask from the most significant bits of each element.
`i movemask_epi8(mi a)`

Extract Bits
≤64 | BMI1 `_bextr` | u32-64
NOTE: Extracts l bits from a starting at bit s.
```
u64 _bextr_u64(u64 a,u32 s,u32 l)
(a >> s) & ((1 << l)-1)
```

Bit Masking

Set or reset a range of bits

Zero High Bits
≤64 | 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
≤64 | 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
≤64 | 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
≤64 | 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 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,ii32 imm)
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,ii imm)
mi sll_epi32(mi a,mi b)
```

Rotate Left/Right
≤64 | x86 `_[l]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
≤64 | 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)`

Bit Counting

Count specific ranges of 0 or 1 bits

Count 1-Bits (Popcount)
≤64 | POPCNT `popcnt` | u32-64
NOTE: Counts the number of 1-bits in a.
`u32 popcnt_u64(u64 a)`

Count Leading Zeros
≤64 | LZCNT `_lzcnt` | u32-64
NOTE: Counts the number of leading zeros in a.
`u64 _lzcnt_u64(u64 a)`

Count Trailing Zeros
≤64 | BMI1 `_tzcnt` | u32-64
NOTE: Counts the number of trailing zeros in a.
`u64 _tzcnt_u64(u64 a)`

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[SSE2]
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 **imm**. For AVX2, there is only si256 and the operation is named inserti128.
```
m insertf128_ps(m a,m b,ii i)
dst[255:0] := a[255:0]
sel := i*128
dst[sel+15:sel]:=b[127:0]
```

Concatenate and Byteshift (Align)
SSSE3 `alignr`
```
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 **permute2x128**.
```
md permute2f128_pd
(md a,md b,ii imm)
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, imm[3:0])
dst[255:128] := S(a, b, imm[7:4])
dst[MAX:256] := 0
```

Interleave (Unpack)
SSE2 `unpackhi/lo` | epi8-64,ps[SSE],pd
NOTE: Interleaves elements from the high/low half of two input registers into a target register.
```
unpackhi_epi32(a,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]
mi unpackhi_epi16(mi a, mi b)
```

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[SSE2]
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!*
```
md shuffle_pd(md a,md b,ii imm)
dst[63:0] := (imm[0] == 0) ?
 a[63:0] : a[127:64]
dst[127:64] := (imm[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 imm)
S(s, 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,imm[1:0])
dst[63:32]:=S(a,imm[3:2])
dst[95:64]:=S(a,imm[5:4])
dst[127:96]:=S(a,imm[7:6])
```

High / Low 16bit Shuffle
SSE2 `shufflehi/lo` | 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 imm)
dst[63:0] := a[63:0]
dst[79:64] :=
 (a >> (imm[1:0]*16))[79:64]
dst[95:80] :=
 (a >> (imm[3:2]*16))[79:64]
dst[111:96] :=
 (a >> (imm[5:4]*16))[79:64]
dst[127:112] :=
 (a >> (imm[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 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 imm)`

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
SSE2 `setzero` | ps[SSE1]/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 lower half of a.
```
md movedup_pd(md a)
a[0:63] | (a[63:0]<<64)
```

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)`

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)`

Byte Movement

Byteshift left/right
SSE2 `[b]sl/rli` | si128,epi128[256]
NOTE: Shift input register left/right by **imm** bytes while shifting in zeros. There is no difference between the **b** and the non-**b** version.
`mi bsrli_si128(mi a,ii imm)`

Byte Swap
≤64 | x86 `_bswap[64]` | (i32,i64)
NOTE: Swaps the bytes in the 32-bit int or 64-bit int.
`i64 _bswap64(i64 a)`

Comparisons

Float Compare

Float Compare
128 | SSE2 `cmp[n]Z` | ps/d,**ss/d**
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. Elements that compare true receive 1s in all bits, otherwise 0s.
`md cmpeq_pd(md a,md b)`

Compare Not NaN
128 | SSE2 `cmp[un]ord` | ps/d,**ss/d**
NOTE: For each element pair cmpord sets the result bits to 1 if **both** elements are not NaN, otherwise 0. cmpunord sets bits if **at least one** is NaN.
`md cmpord_pd(md a,md b)`

Compare
AVX `cmp` | ps/d,ss/d
NOTE: Compares packed or single elements based on the third operand which is an **immediate**. Possible values are 0-31. Elements that compare true receive 1s in all bits, otherwise 0s. Check documentation for semantics.
`md cmp_pd(md a,md b,ii imm)`

Compare Single Float
SSE2 `[u]comiZ` | **ss/d**
NOTE: Z can be one of: eq/ge/gt/le/lt/neq
Returns a single int that is either 1 or 0. The **u** version does not signal an exception for QNaNs and is not available for 256 bits!
`i comieq_sd(mi a,mi b)`

Int Compare

Int Compare
SSE2 `cmpZ` | epi8-32,epi64[SSE4.1]
NOTE: Z can be one of: lt/gt/eq. Elements that equal receive 1s in all bits, otherwise 0s.
`mi cmpeq_epi8(mi a,mi b)`

Bit Compare

Perform a bitwise operation and check whether all bits are 0s afterwards

Test And Not/ And
SSE4.1, AVX `testc/z` | si128[SSE4.1],si256, ps/d
NOTE: Compute the bitwise AND of 128 bits (representing integer data) in a and b, and set ZF to 1 if the result is zero, otherwise set ZF to 0. Compute the bitwise AND NOT of a and b, and set CF to 1 if the result is zero, otherwise set CF to 0. The **c** version returns CF and the **z** version ZF. For 128 bit, there is also the operation test_all_zeros which does the same as testz_si128.
```
i testc_si128(mi a,mi b)
ZF := ((a & b) == 0)
CF := ((a & !b) == 0)
RETURN CF;
```

Test Mix Ones Zeros
SSE4.1, AVX `test_ncz` | si128[SSE4.1],si256 ,ps/d
NOTE: Compute the bitwise AND of 128 bits (representing integer data) in a and b, and set ZF to 1 if the result is zero, otherwise set ZF to 0. Compute the bitwise AND NOT of a and b, and set CF to 1 if the result is zero, otherwise set CF to 0. Return !CF && !ZF. For 128 bit, there is also the operation test_mix_ones_zeros which does the same.
```
i test_mix_ones_zeros(mi a,mi b)
ZF := ((a & b) == 0)
CF := ((a & !b) == 0)
RETURN !CF && !ZF;
```

Test All Ones
SEQ! | 128 | SSE4.1 `test_all_ones` | -
NOTE: Returns true iff all bits in a are set. *Needs two instructions and may be slower than native implementations.*
```
i test_all_ones(mi a)
(~a) == 0
```

String Compare

String Compare Description
```
cmpistrX(mi a, mi b, ii imm)
cmpestrX(mi a, i la, mi b, i lb, ii imm)
```
Each operation has an **i** and an **e** version. The **i** version compares all elements, the **e** version compares up to specific lengths **la** and **lb**. The **immediate** value for all these comparisons consists of bit flags. Exactly one flag per group must be present:

Data type specifier	
_SIDD_UBYTE_OPS	unsigned 8-bit chars
_SIDD_UWORD_OPS	unsigned 16-bit chars
_SIDD_SBYTE_OPS	signed 8-bit chars
_SIDD_SWORD_OPS	signed 16-bit chars

Compare mode specifier	
_SIDD_CMP_EQUAL_ANY	For each character c in a, determine iff any character in b is equal to c.
_SIDD_CMP_RANGES	For each character c in a, determine whether b0 <= c <= b1or b2 <= c <= b3…
_SIDD_CMP_EQUAL_ORDERED	Check for string equality of a and b
_SIDD_CMP_EQUAL_EACH	Search substring b in a. Each byte where b begins in a is treated as match.

Polarity specifier	
_SIDD_POSITIVE_POLARITY	Match is indicated by a 1-bit.
_SIDD_NEGATIVE_POLARITY	Negation of resulting bitmask.
_SIDD_MASKED_NEGATIVE_POLARITY	Negation of resulting bitmask except for bits that have an index larger than the size of a or b.

String Compare Mask
SSE4.2 `cmpi/estrm` | ps/d,ss/d
NOTE: Compares strings in **a** and **b** and returns the comparsion mask. If _SIDD_BIT_MASK is used, then the resulting mask is a bit mask. If _SIDD_UNIT_MASK is used, then the result is a byte mask which has ones in all bits of the bytes that do not match.
```
mi cmpistrm(mi a,mi b,ii imm)
mi cmpestrm
(mi a,i la,mi b,i lb,ii imm)
```

String Compare Index
SSE4.2 `cmpi/estri` | ps/d,ss/d
NOTE: Compares strings in **a** and **b** and returns the index of the first byte that matches. Otherwise Maxsize is returned (either 8 or 16 depending on data type).
```
i cmpistri(mi a,mi b,ii imm)
i cmpestri
(mi a,i la,mi b,i lb,ii imm)
```

String Compare
SSE4.2 `cmpi/estrc` | ps/d,ss/d
NOTE: Compares strings in **a** and **b** and returns true iff the resulting mask is not zero, i.e., if there was a match.
```
i cmpistrc(mi a,mi b,ii imm)
i cmpestrc
(mi a,i la,mi b,i lb,ii imm)
```

String Compare with Nullcheck
SSE4.2 `cmpi/estra` | ps/d,ss/d
NOTE: Compares strings in **a** and **b** and returns true iff the resulting mask is zero and there is no null character in **b**.
```
i cmpistra(mi a,mi b,ii imm)
i cmpestra
(mi a,i la,mi b,i lb,ii imm)
```

String Nullcheck
SSE4.2 `cmpi/estrs/z` | ps/d,ss/d
NOTE: Compares two strings a and b and returns if **a** (**s** version) or **b** (**z** version) contains a null character.
```
i cmpistrs(mi a,mi b,ii imm)
i cmpestrs
(mi a,i la,mi b,i lb,ii imm)
```

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 sixteen-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 with correct rounding.
`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)`

Add
SSE, SSE2 `add` | epi8-64,ps[SSE]/d, **ss[SSE]/d**
`mi add_epi16(mi a,mi b)`

Add with Saturation
SSE, SSE2 `adds` | epi8-16,epu8-16
`mi adds_epi16(mi a,mi b)`

Alternating Add and Substract
SSE3 `addsub` | ps/d
```
m addsub_ps(m a,m b)
FOR j := 0 to 3
 i := j*32
 IF (j is even)
  dst[i+31:i] :=
   a[i+31:i] - b[i+31:i]
 ELSE
  dst[i+31:i] :=
   a[i+31:i] + b[i+31:i]
```

Horizontal Substract with Saturation
SSSE3 `hsubs` | epi16
NOTE: Substracts adjacent pairs of elements with saturation
`mi hsubs_epi16(mi a,mi b)`

Horizontal Substract
SSSE3 `hsub` | epi16-32,ps/d
NOTE: Substracts adjacent pairs of elements
`mi hsub_epi16(mi a,mi b)`

Subtract
SSE, SSE2 `sub` | epi8-64,ps[SSE]/d, **ss[SSE]/d**
`mi sub_epi16(mi a,mi b)`

Subtract with Saturation
SSE2 `subs` | epi8-16,epu8-16
`mi subs_epi16(mi a,mi b)`

Div/Sqrt/Reciprocal

Div
SSE2 `div` | ps/d,**ss/d**
`m div_ps(m a,m b)`

Approx. Reciprocal
SSE `rcp` | ps,ss
`m rcp_ps(m a)`

Approx. Reciprocal Sqrt
SSE `rsqrt` | ps,ss
NOTE: Approximates 1.0/sqrt(x)
`m rsqrt_ps(m a)`

Square Root
SSE, SSE2 `sqrt` | ps/d,**ss/d**
`m sqrt_ps(m a)`

Min/Max/Avg

Horizontal Min
128 | SSE4.1 `minpos` | epu16
NOTE: Computes horizontal min of one input vector of 16bit uints. The min is stored in the lower 16 bits and its index is stored in the following 3 bits
`mi minpos_epu16(mi a)`

Min
SSE2, SSE4.1 `min` | **SSE:**ps **SSE2:**epu8, epi16,pd **SSE4.1:** epi8-32,epu8-32
`mi min_epu16(mi a,mi b)`

Max
SSE2, SSE4.1 `max` | **SSE:**ps **SSE2:**epu8, epi16,pd **SSE4.1:** epi8-32,epu8-32
`mi max_epu16(mi a,mi b)`

Average
SSE2 `avg` | epu8-16
`mi avg_epu16(mi a,mi b)`

Sign Modification

Absolute
SSSE3 `abs` | epi8-32
`mi abs_epi16(mi a)`

Conditional Negate or Zero
SSSE3 `sign` | epi8-32
NOTE: For each element in **a** and **b**, set result element to **a** if b is positive, set result element to **-a** if **b** is negative or set result element to 0 if b is 0.
`mi sign_epi16(mi a,mi b)`

Composite Arithmetics

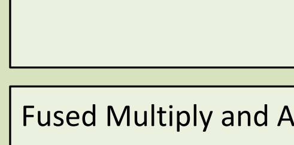
Perform more than one operation at once

Dot Product

Conditional Float Dot Product
SSE4.1 `dp` | ps/d
NOTE: Computes the dot product of two vectors. A mask is used to state which components are to be multiplied and stored.
```
m dp_ps(m a,m b,ii imm)
FOR j := 0 to 3
 IF imm[4+j]
  tmp[i+31:i] :=
   a[i+31:i] * b[i+31:i]
 ELSE
  tmp[i+31:i] := 0
sum[31:0] :=
tmp[127:96] + tmp[95:64]
+ tmp[63:32]+ tmp[31:0]
FOR j := 0 to 3
 IF imm[j]
  dst[i+31:i] :=sum[31:0]
 ELSE
  dst[i+31:i] := 0
```

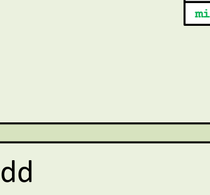
Composite Int Arithmetics

Byte Multiply and Horizontal Saturated Add
SSSE3 `maddubs` | epi16
NOTE: Adds vertical 8 bit ints producing a 16 bit int and adds horizontal pairs with saturation producing 8x16 bit ints. *The first input is treated as unsigned and the second as signed. Results and intermediaries are signed.*



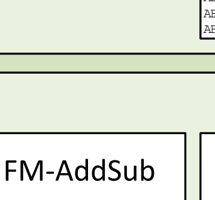
`mi maddubs_epi16(mi a,mi b)`

Multiply and Horizontal Saturated Add
SSE3 `madd` | epi16
NOTE: Multiply vertical 16 bit ints producing a 32 bit int and adds horizontal pairs with saturation producing 4x32 bit ints



`mi madd_epi16(mi a,mi b)`

Sum of Absolute Differences
SSE2 `sad` | epu8
NOTE: Compute the absolute differences of packed unsigned 8-bit integers in a and b, then horizontally sum each consecutive 8 differences to produce two unsigned 16-bit integers, and pack these unsigned 16-bit integers in the low 16 bits of 64-bit elements in dst.



`mi sad_epu8(mi a,mi b)`

Sum of Absolute Differences 2
SSE4.1 `sadbw` | epu8
NOTE: Compute the sum of absolute differences (SADs) of quadruplets of unsigned 8-bit integers in a compared to those in b, and store the 16-bit results in dst. Eight SADs are performed using one quadruplet from b and eight quadruplets from a. One quadruplet is selected from b starting at on the offset specified in imm. Eight quadruplets are formed from sequential 8-bit integers selected from a starting at the offset specified in imm.
```
mi sadbw_epu8(mi a,mi b,ii imm)
{
 a_offset := imm[2]*32
 b_offset := imm[1:0]*32
 FOR j := 0 to 7
  i := j*8
  k := a_offset+i
  l := b_offset
  dst[i+15:i] :=
ABS(a[k+7:k]-b[l+7:l])+
ABS(a[k+15:k+8]-[l+15:l+8])+
ABS(a[k+23:k+16]-b[l+23:l+16])+
ABS(a[k+31:k+24]-[l+31:l+24])
```

Fused Multiply and Add

FM-Add
FMA `f[n]madd` | ps/d,**ss/d**
NOTE: Computes (a*b)+c for each element. The **n** version computes -(a*b)+c.
`m fmadd_ps(m a,m b,m c)`

FM-Sub
FMA `f[n]madd` | ps/d,**ss/d**
NOTE: Computes (a*b)-c for each element. The **n** version computes -(a*b)-c.
`m fmsub_ps(m a,m b,m c)`

FM-AddSub
FMA `fmaddsub` | ps/d,**ss/d**
NOTE: Computes (a*b)+c for elements with even index and (a*b)-c for elements with odd index.

FM-SubAdd
FMA `fmsubadd` | ps/d,**ss/d**
NOTE: Computes (a*b)-c for elements with even index and (a*b)+c for elements with odd index.
`m fmsubadd_ps(m a,m b,m c)`

Version 1.0