Vector SIMD Instructions

Contents

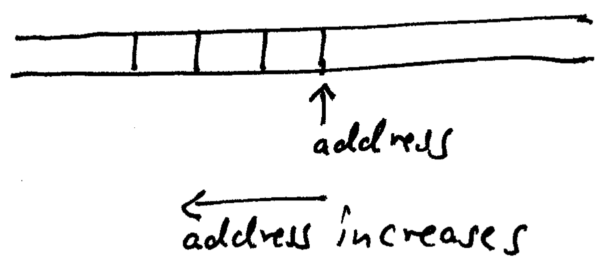
Information

vector indexing: a:

a_3	a_2	a_1	a_0
3	2	1	0

in most intrinsics, the order of operands matter

memory:



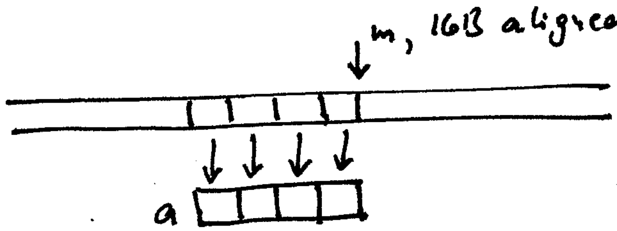
SIMD extensions time line: SSE, SSE2, SSE3, SSSE3, SSE4

We focus on single precision float, 4-way

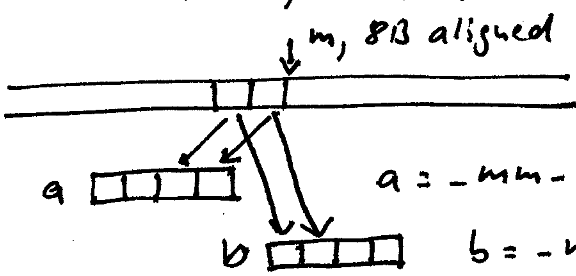
1 vector = 128 bit = 16 B, data type __m128

Unless stated otherwise, instructions are SSE or later

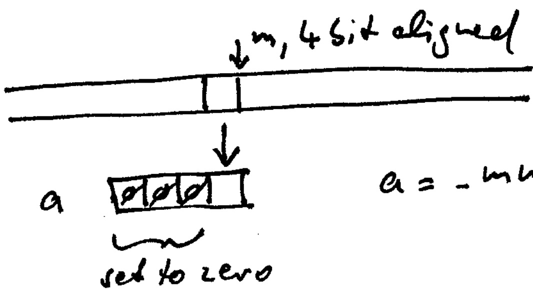
Load and Store




```
a = _mm_load_ps(m);      aligned
a = _mm_loadu_ps(m);     unaligned (avoid)
a = p[i];  if p is:  __m128 *p
```




```
a = _mm_loadl_pi(a, m);   (keeps upper half)
b = _mm_loadh_pi(b, m);   (keeps lower half)
```




```
a = _mm_load_ss(m)
```

stores are analogous

Constants

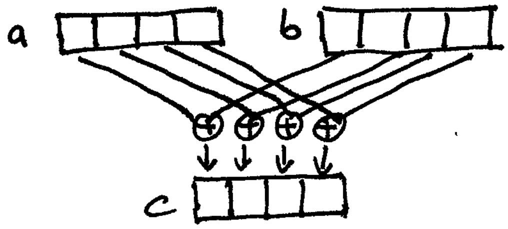
c: | 4.0 | 3.0 | 2.0 | 1.0 | `c = _mm_set_ps(4.0, 3.0, 2.0, 1.0);`

d: | 1.0 | 1.0 | 1.0 | 1.0 | `d = _mm_set1_ps(1.0);`

e: | Ø | Ø | Ø | 1.0 | `e = _mm_set_ss(1.0);`

f: | Ø | Ø | Ø | Ø | `f = _mm_setzero_ps();`

Vector arithmetic



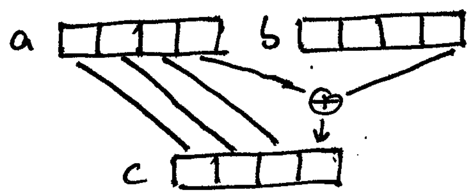
c = _mm_add_ps(a, b); "a + b"

analogous:

c = _mm_sub_ps(a, b); "a − b"

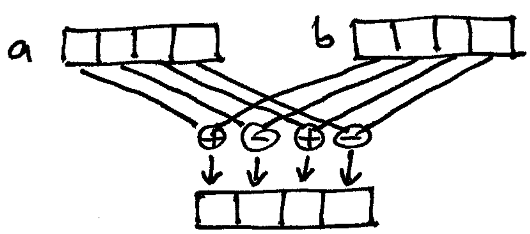
c = _mm_mul_ps(a, b); "a · b"

Scalar arithmetic

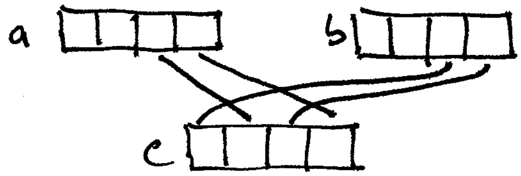


c = _mm_addss(a, b);

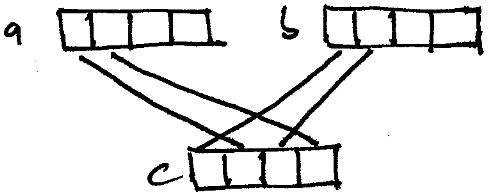
AddSub (SSE3 and later)



Reorder Instructions

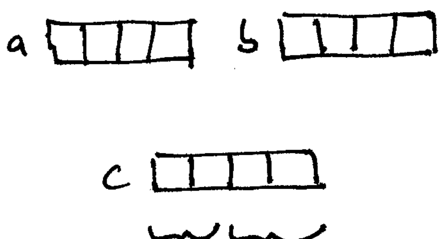


$c = $ _mm_unpacklo_ps(a, b)



$c = $ _mm_unpackhi_ps(a, b)

shuffle:



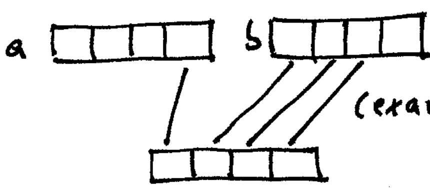
any element of b   any element of a

$c = $ _mm_shuffle_ps(a, b, _MM_SHUFFLE(l, k, j, i))

$c_0 = a_i$
$c_1 = a_j$
$c_2 = b_k$
$c_3 = b_l$

$i, j, k, l \in \{0..3\}$

align: (SSE3 and later)

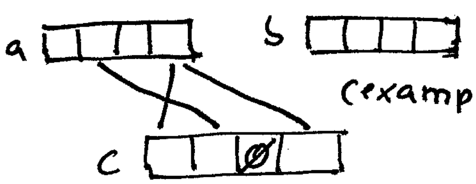


(example)

"any 4 consecutive elements of the concatenation of a and b go into c"

_mm_alignr_epi8   use with _mm_castsi128_ps

shuffle: (SSE3 and later)

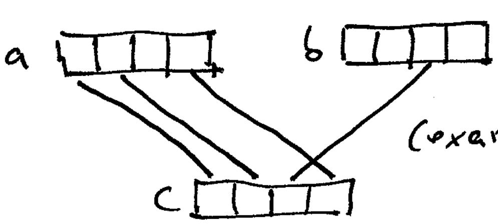


(example)

"c is filled in each position with any element from a or 0, as specified by b"

_mm_shuffle_epi8

blend: (SSE4 and later)

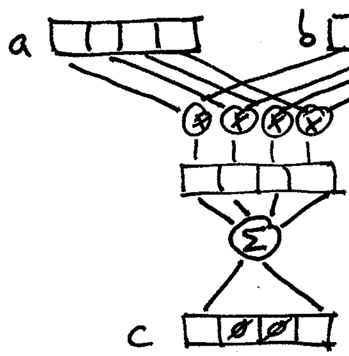


(example)

"c is filled in each position with an element from a or b from the same position"

_mm_blend_ps

Dot product (SSE4 and later)

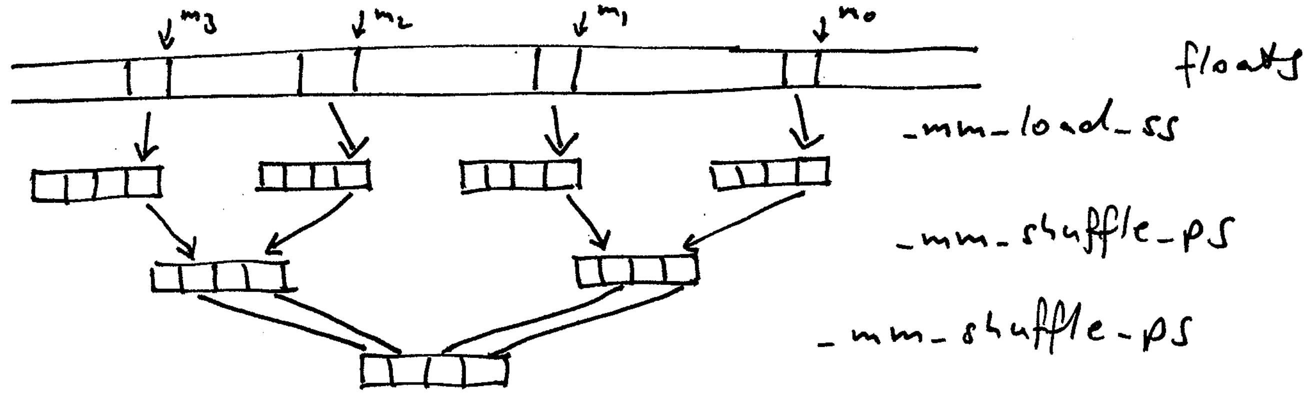


(example)

"computes the pointwise product of a and b and writes an arbitrary sum of the resulting numbers into selected elements of c — the others are set to zero"

_mm_dp_ps(a, b, mask)

Load 4 real numbers from arbitrary memory locations



7 instructions, this is the right way

Note:
- wherever possible avoid this by restructuring the algorithm or data to have aligned vector loads _mm_load_ps
- the above should be equivalent to the following but a.) the above is safer; b.) be aware that the below are 7 instructions

```
float f[20] = {...};
__m128 vf = _mm_set_ps(f[3], f[5], f[17], f[1]);
```

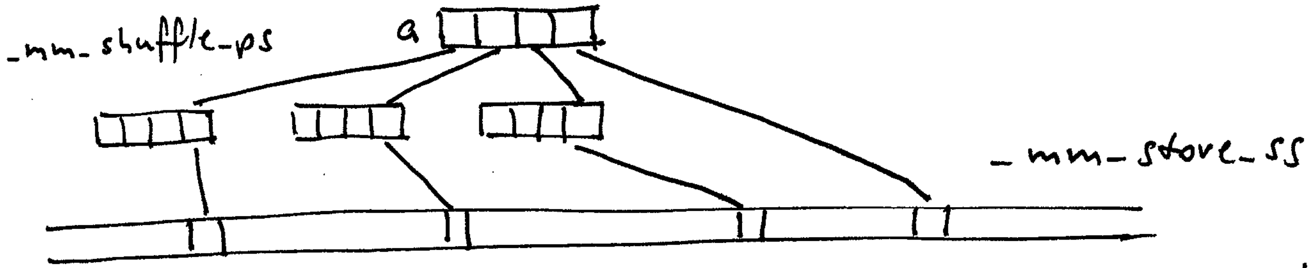
Don't do this:

```
float f[20] = {...};
__declspec(align(16)) g[4];
__m192 vf;
g[0] = f[1];
g[1] = f[17];
g[2] = f[5];
g[3] = f[3];
vf = _mm_load_ps(g);
```

$\left.\right\}$ mem → register → mem round trip ⇒ expensive

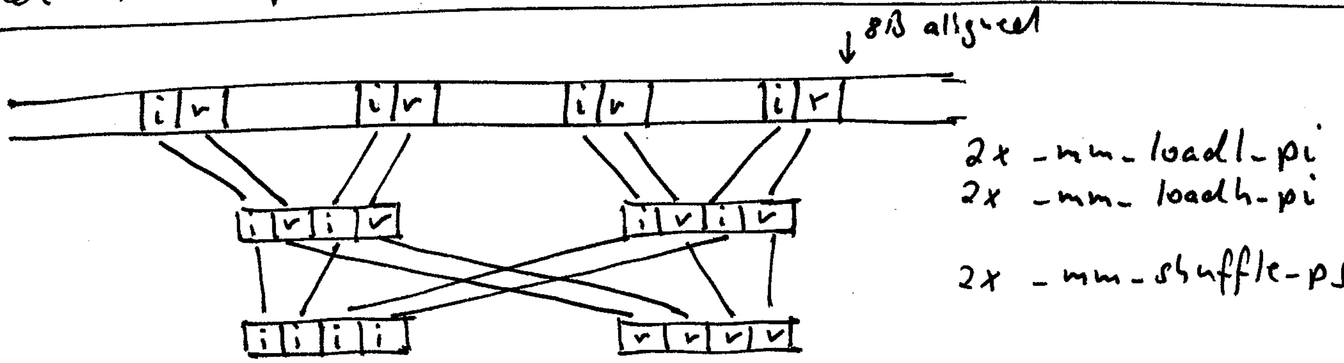
(same problem with unions and pointers)

Store 4 reel numbers to arbitrary memory locations

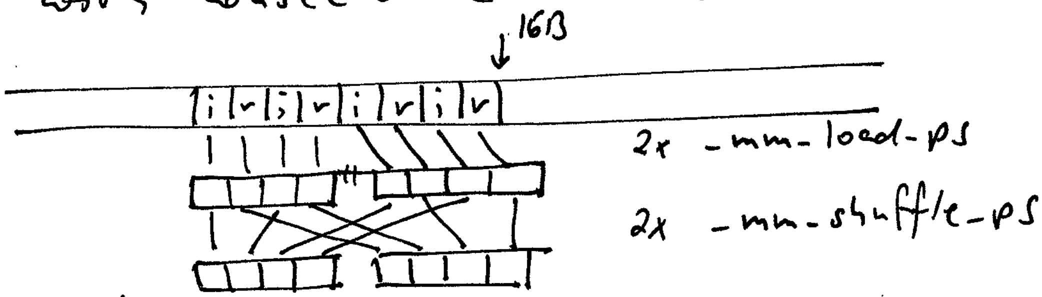


7 instructions, shorter critical path than load

Load 4 complex numbers (= 4 pairs of reel numbers)

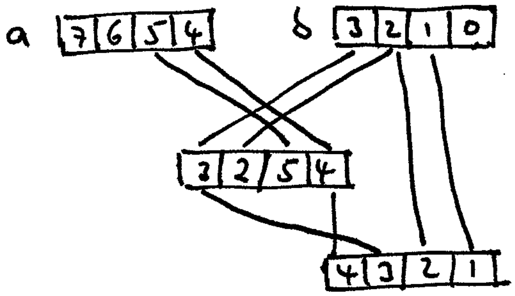


6 instructions
store analogous
same with consecutive data:



4 instructions
store analogous

Shift by 1



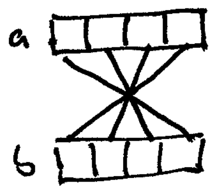
_mm_shuffle_ps

_mm_shuffle_ps

2 instructions

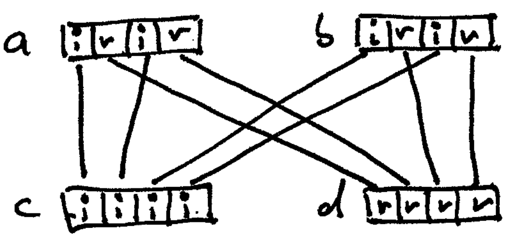
SSE3 and later: _mm_alignr_epi8 + casts   1 instruction

Reverse vector



b = _mm_shuffle_ps(a, a, _MM_SHUFFLE(0,1,2,3));

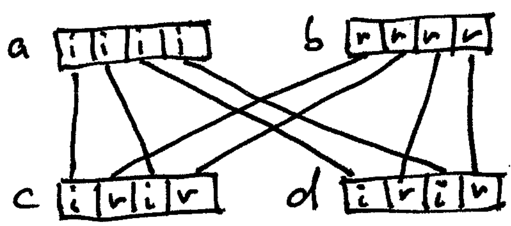
Interleaved complex → split complex



c = _mm_shuffle_ps(b, a, _MM_SHUFFLE(3,1,3,1));

d = ... _MM_SHUFFLE(2,0,2,0));

Split complex → interleaved complex



c = _mm_unpackhi_ps(b, a);

d = _mm_unpacklo_ps(b, a);

Transposition: 4x4 matrix

4x4 matrix: $\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} = A$

in memory:

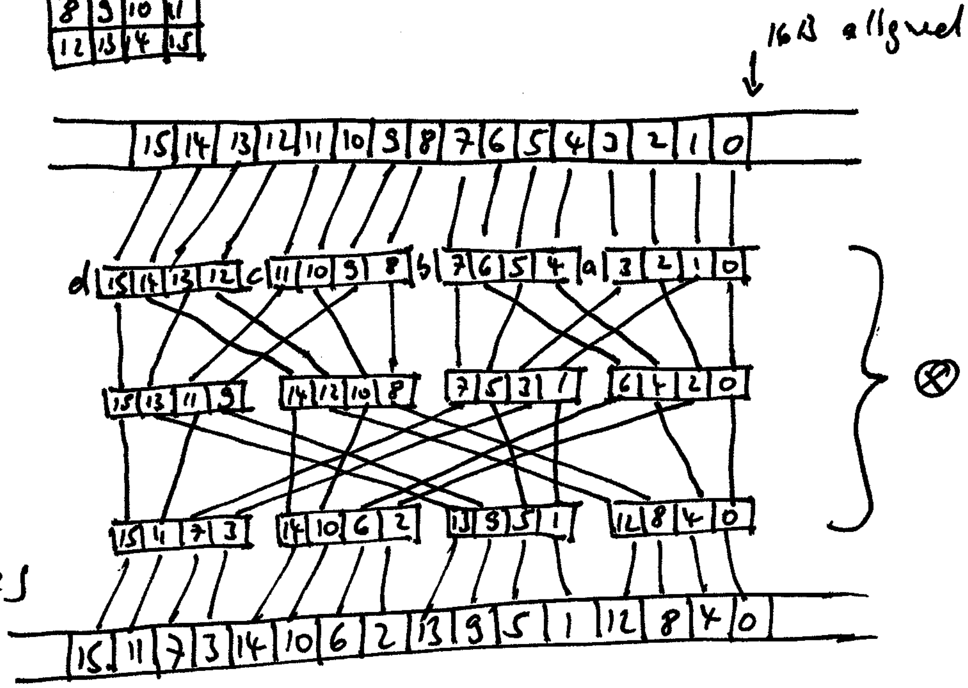
4 aligned loads

4 shuffles

4 shuffles

4 aligned stores

in memory:

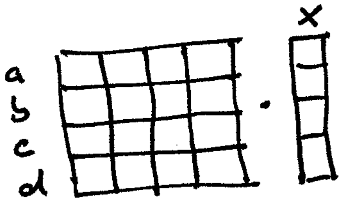


as matrix: $\begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix} = A^T$

⊗ done by the macro `_MM_TRANSPOSE4_PS(a,b,c,d);`

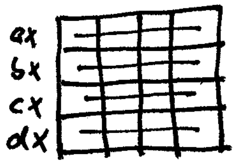
8 instructions

Matrix - vector product



1. step: 4 vector products ax, bx, cx, dx (4 instructions)

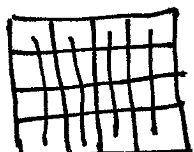
result:



SSE:

2. step: transpose (8 instructions)

result:

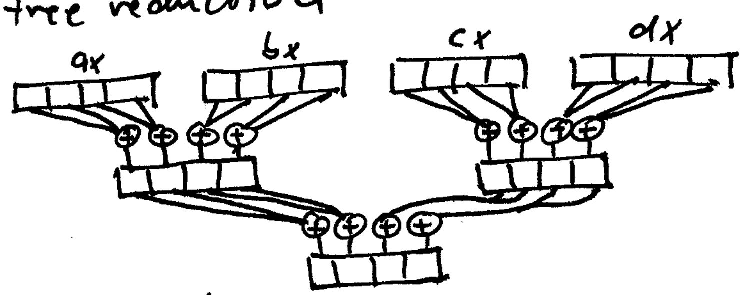


3. step: sum rows (3 instructions)

total: 15 instructions

SSE3:

2. step: tree reduction



3 instructions (_mm_hadd_ps)

total: 7 instructions

SSE4: has dot product instruction but still 7 instructions are needed (exercise)