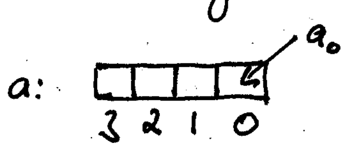
Table of Contents

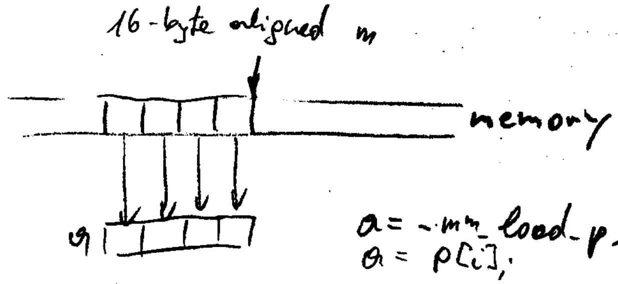
Instructions: 1–7
Basic building blocks:
- loads & stores 8–12
- shuffles 13–15
- arithmetic 16–18

vector indexing:

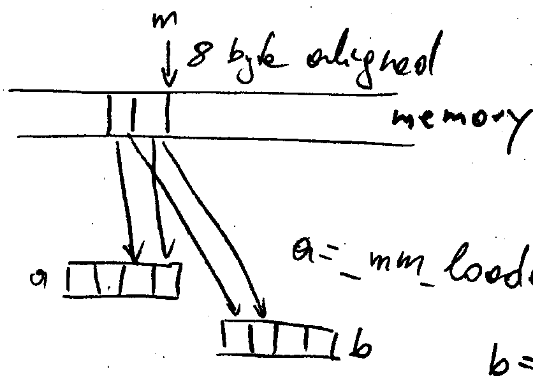


in most intrinsics the order of operands matter!

Load Instructions (SSE and later)

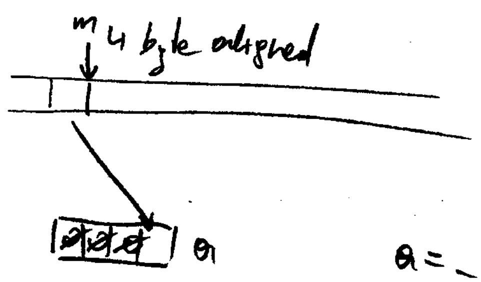


a = _mm_load_ps(m);   aligned, explicit
a = p[i];   — " —, implicit
a = _mm_loadu_ps(m);   unaligned



a = _mm_loadl_pi(a, m)

b = _mm_loadh_pi(b, m)



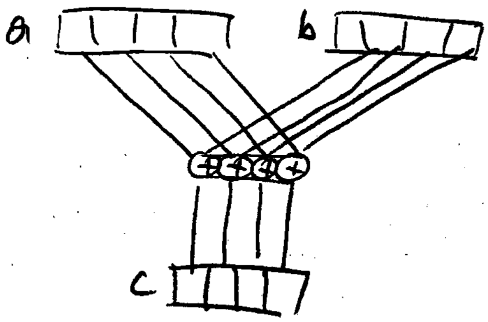
a = _mm_load_ss(m)

$\boxed{\text{stores are analogous}}$

Constants (Special Load instructions) (SSE and later)

c = _mm_set_ps(1.0, 2.0, 3.0, 4.0);	4.0 | 3.0 | 2.0 | 1.0
d = _mm_set1_ps(1.0);	1.0 | 1.0 | 1.0 | 1.0
e = _mm_set_ss(1.0);	0 | 0 | 0 | 1.0
f = _mm_setzero_ps();	0 | 0 | 0 | 0

Vector arithmetic (SSE and later)



c = _mm_add_ps(a, b)

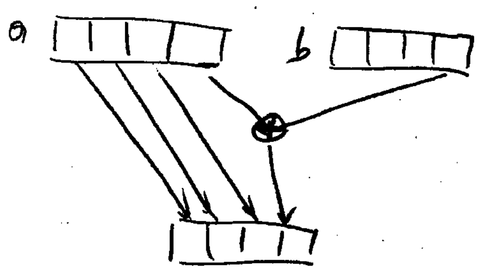
Same:

c = _mm_sub_ps(a, b)

c = _mm_mul_ps(a, b)

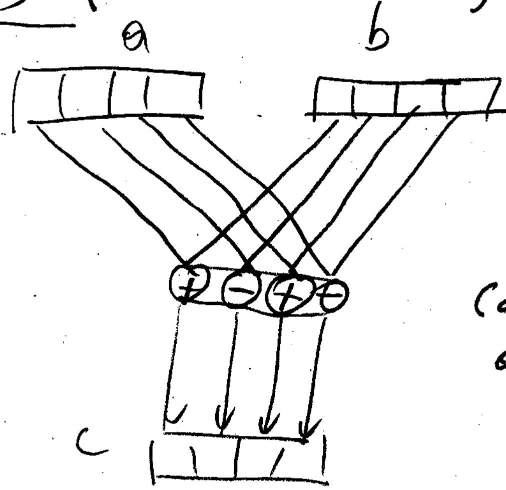
⋮

Scalar arithmetic (SSE and later)



c = _mm_add_ss(a, b)

Add Sub (SSE3 and later)



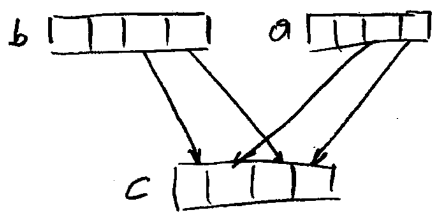
c = _mm_addsub_ps(a, b)

(alternate add & sub of vector elements)

Reorder Instructions (SSE and later)

Unpack lo

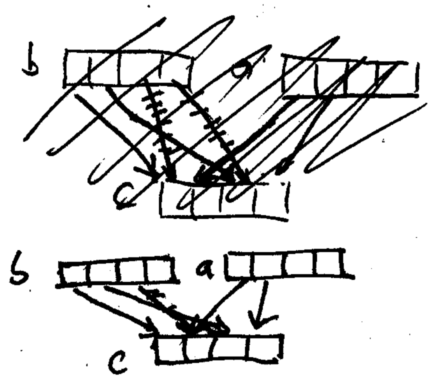
$c = \_mm\_unpacklo\_ps(a,b)$



$c_0 = a_0$
$c_1 = b_0$
$c_2 = a_1$
$c_3 = b_1$

Unpack hi

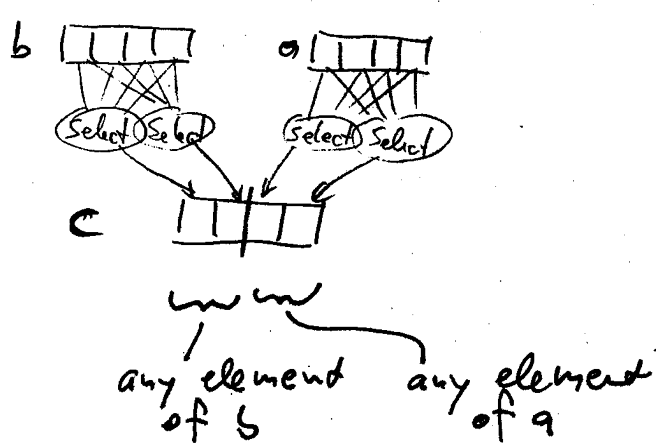
$c = \_mm\_unpackhi\_ps(a,b)$



$c_0 = a_2$
$c_1 = b_2$
$c_2 = a_3$
$c_3 = b_3$

Shuffle

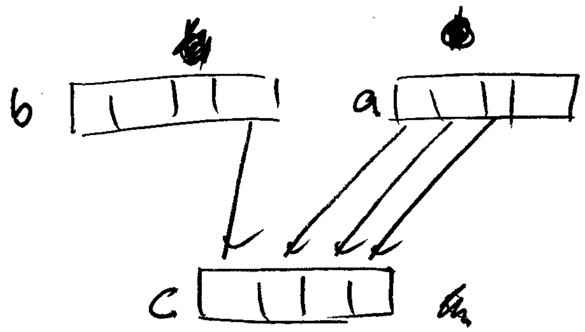
$c = \_mm\_shuffle\_ps(a,b,\_MM\_SHUFFLE(l,k,j,i))$



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

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

align (SSSE3 and later)

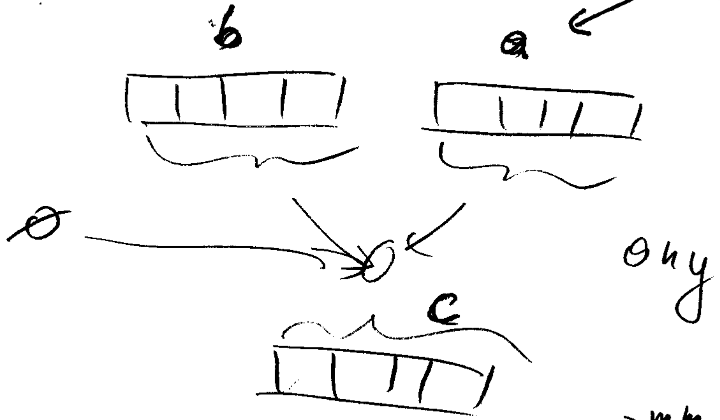


"any consecutive 4 elements of the concatenation of a and b goes into c"

```
c=_mm_castps_si128(_mm_alignr_epi8(
    _mm_castsi128_ps(a)
    _mm_castsi128_ps(b)));
```

shuffle (SSSE3 and later)

can be given at runtime

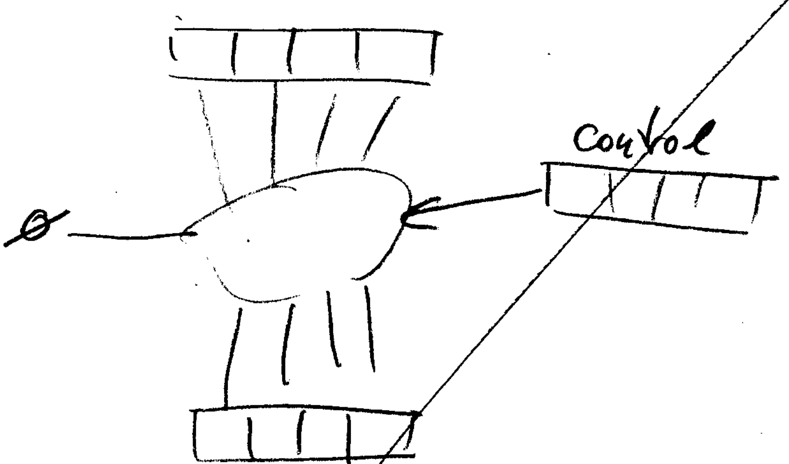


"c is filled with any element of b or 0"
(specified by a)

```
_mm_shuffle_epi8( )
```

shuffle (SSSE3 and later)

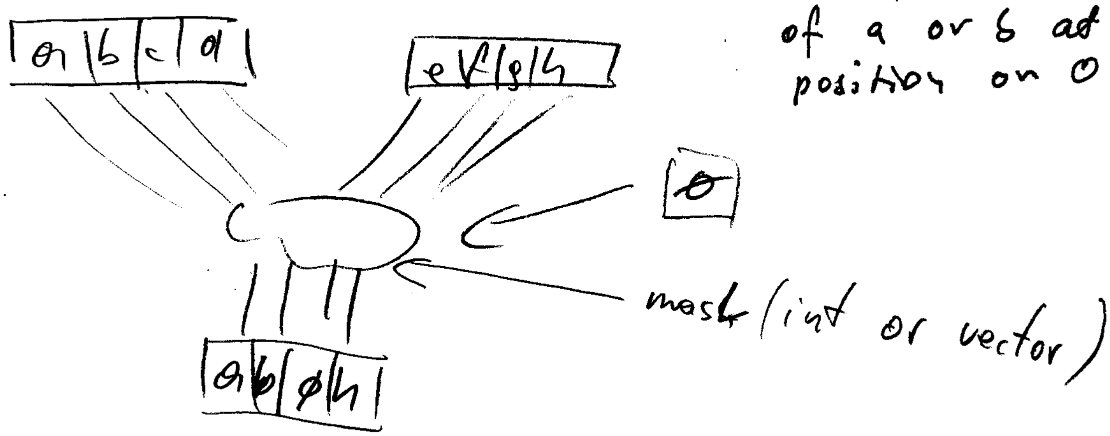
- wrong last time
- correct



- not the same as AltiVec

Blend instruction

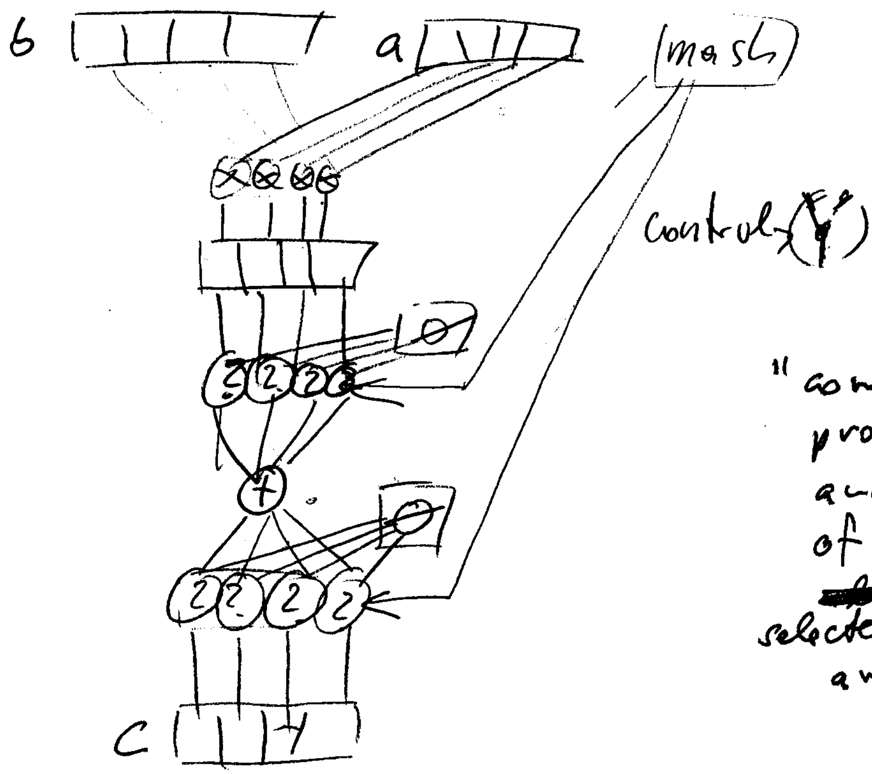
"c is filled with elements of a or b at the same position on 0"



Matrix-Vector Product (SSE4 and later

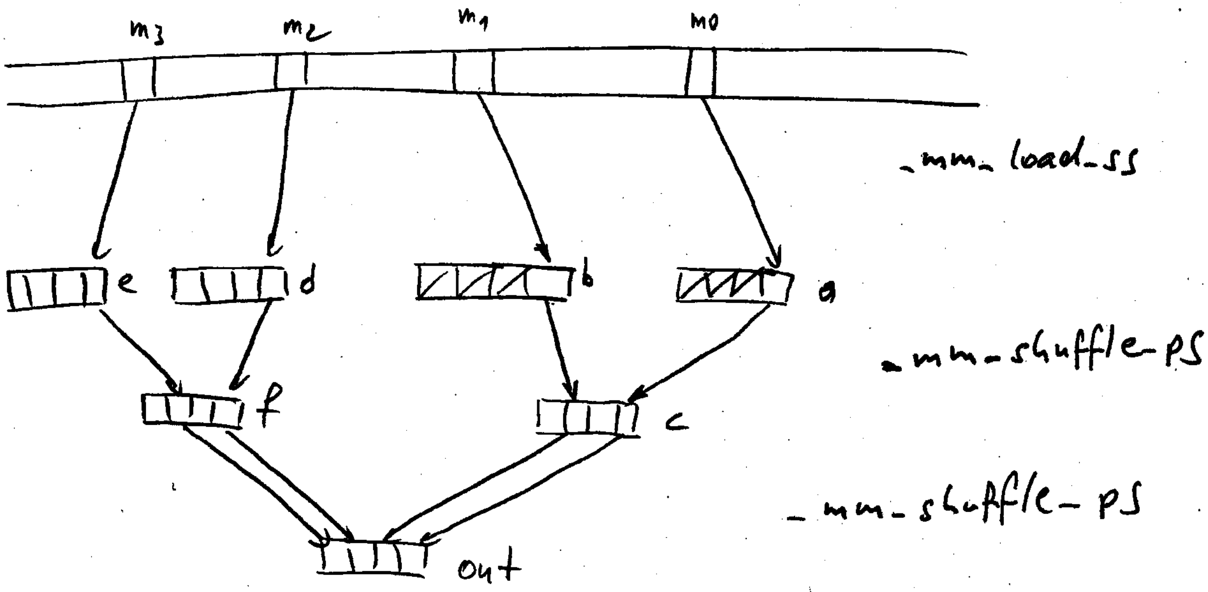
Dot-product instruction

_mm_dp_ps(a, b, mask) ← magic



"computes the pointwise product of a and b and writes an arbitrary sum of the resulting numbers into ~~all~~ selected elements of c (the others are set to 0"

Example: load 4 real numbers from arbitrary memory locations (SSE)




```
#define SCALAR_LOAD (out, m0, m1, m2, m3)    { \
    a = _mm_load_ss(m0);                          \
    b = _mm_load_ss(m1);                          \
    c = _mm_shuffle_ps(a, b, _MM_SHUFFLE(1,0,1,0)); \
    d = _mm_load_ss(m2);                          \
    e = _mm_load_ss(m3);                          \
    f = _mm_shuffle_ps(d, e, _MM_SHUFFLE(1,0,1,0)); \
  out = _mm_shuffle_ps(c, f, _MM_SHUFFLE(1,0,1,0)); \
}
```

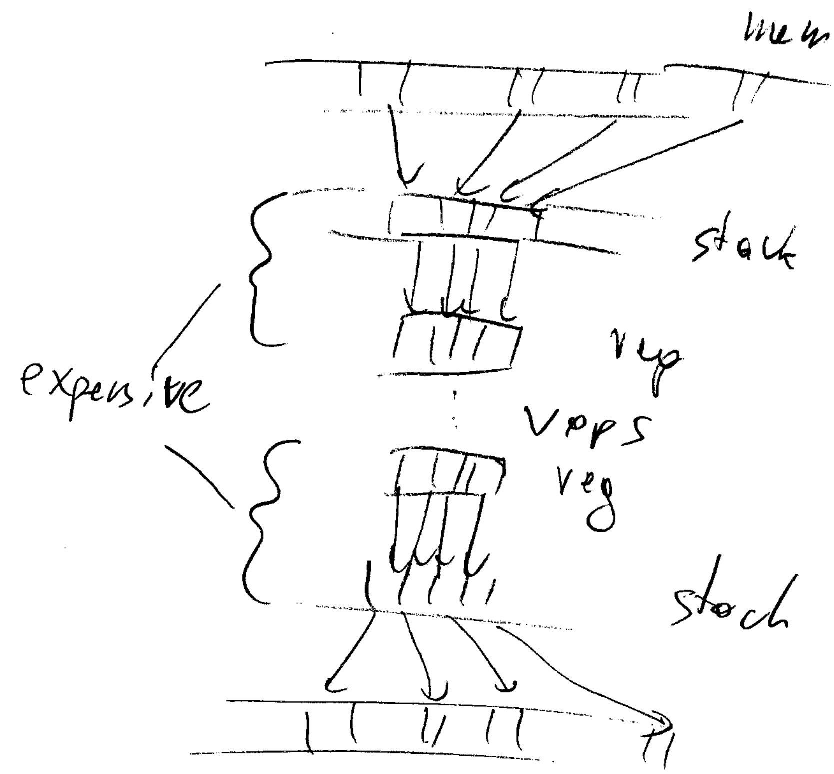
7 instructions

Note: - Whenever possible avoid this by ~~aligning~~ restructuring the algorithm or data format to have aligned vector loads (see page 1)

- equivalent to macro on page 11 (but the above is "safer")

Other Gather/Scatter implementation (BAD) Don't do it like this


```
float f[20] = {...};
__declspec(align(16)) g[4];
__m128 vf;
 g[0] = f[3];
 g[1] = f[5];
 g[2] = f[12];
 g[3] = f[17];
 vf = _mm_loadps(g);
 // operations
 _mm_storeps(g, vf);
 f[7] = g[0];
 f[13] = g[1];
 f[11] = g[2];
 f[17] = g[3];
```



looks innocent, but is really bad
most people try that at some point
look at the assembly to see for yourself

Set Instruction ~~tricks and wrong~~ (SSE and later)

You can do : (see page 1)

```
__m128 vf = _mm_set_ps(4.0, 3.0, 2.0, 1.0);
```

→ 1 vector load of 128 bit constant

Compiler lets you do ~~this type of use~~

```
float f[20] = {...};
__m128 vf = _mm_set_ps( f[3], f[5], f[12], f[17]);
```

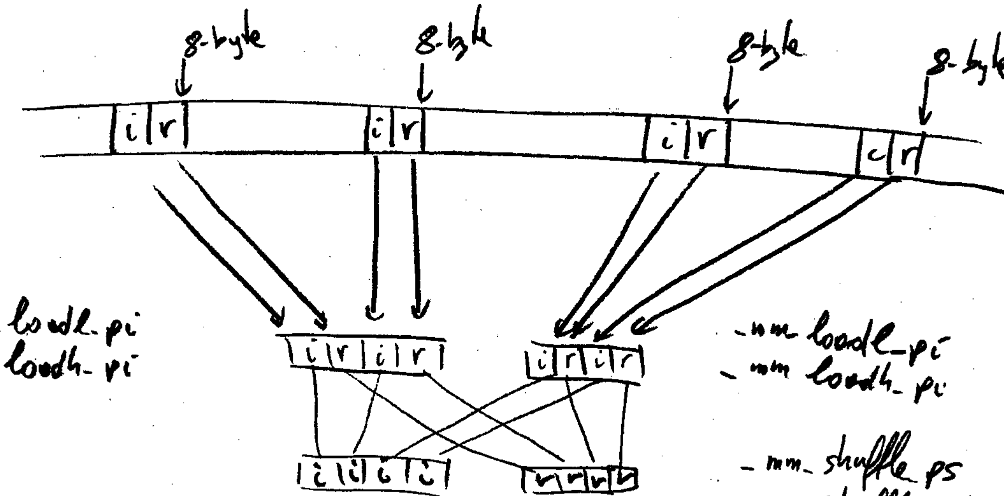
however internally : 4 loads, 3 shuffles equivalent to page 8

Do not use _mm_set_ps() on variables if you can avoid it! (see page 8)

Example: load 4 complex numbers (load 4 pairs of numbers)

(SSE and later)

Strided access



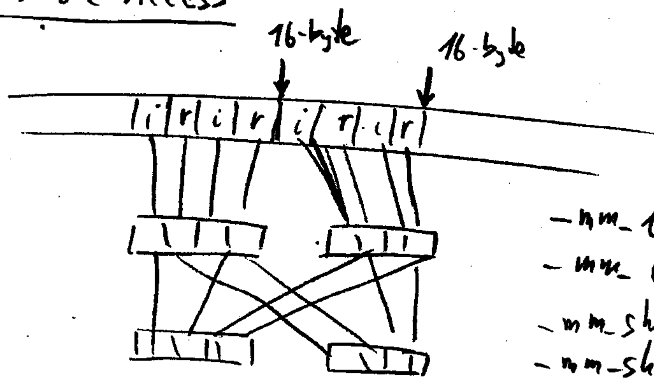
_mm_loadl_pi
_mm_loadh_pi

_mm_loadl_pi
_mm_loadh_pi

_mm_shuffle_ps
_mm_shuffle_ps

6 instructions

Unit stride Access



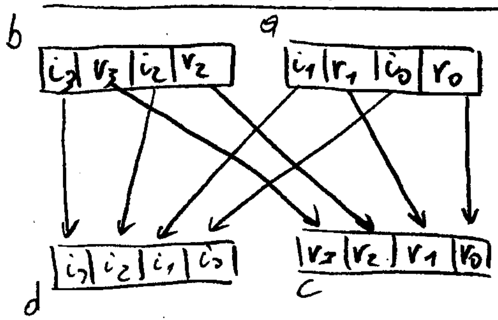
_mm_load_ps
_mm_load_ps
_mm_shuffle_ps
_mm_shuffle_ps

4 instructions
(benefit of consecutive data)

Same for store ops

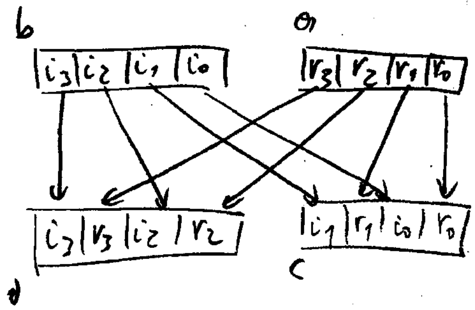
Reorder Instructions – Examples (SSE and later)

Interleaved complex → split complex : L^8_2



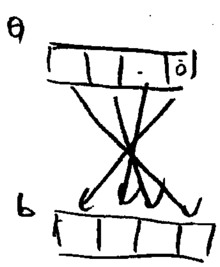
```
c = _mm_shuffle_ps(a, b, _MM_SHUFFLE(2,0,2,0));
d = _mm_shuffle_ps(a, b, _MM_SHUFFLE(3,1,3,1));
```

Split complex → interleaved complex : L^8_4



```
c = _mm_unpacklo_ps(a, b);
d = _mm_unpackhi_ps(a, b);
```

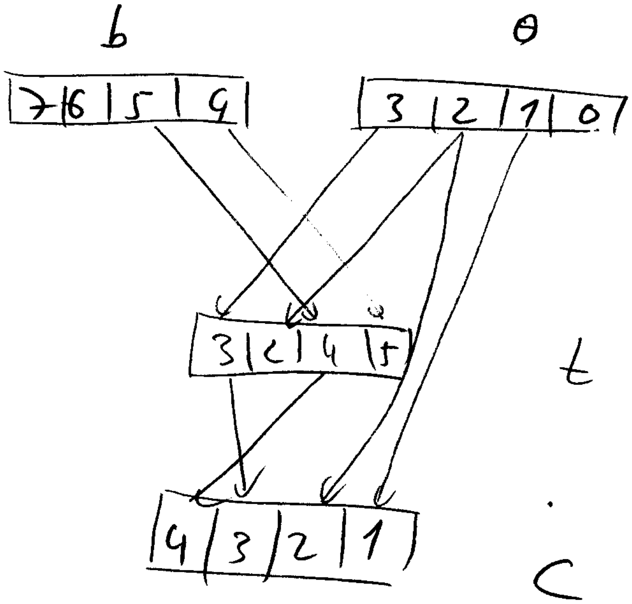
Reverse Vector : J_4



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

~~Shift by~~

Shift by 1 (SSE)



2 instructions

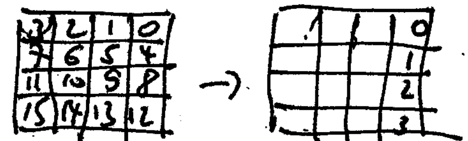
t = _mm_shuffle_ps(b, a, _MM_SHUFFLE(3, 2, 1, 0));

c = _mm_shuffle_ps(, , _MM_SHUFFLE(1, 3, 2, 1));

SSSE3: palignr _mm_alignr_epi8() 1 instruction

~~pshufb _mm_shuffle_epi8()~~

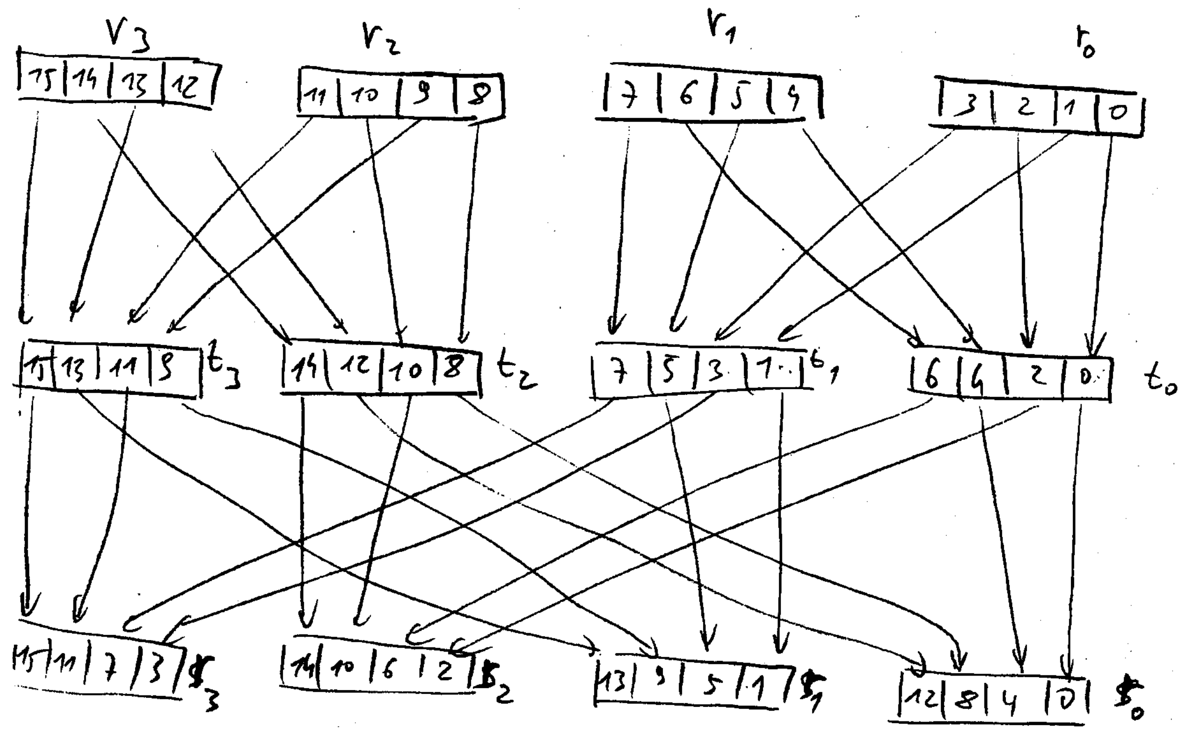
Reorder Instructions - Examples



vectors are rows

may need fixing

Transpose 4x4 matrix: $\cancel{L^{16}_4 = (L^4_2 \otimes I_4)(I_2 \otimes L^8_2)(L^4_2 \otimes I_4)(I_2 \otimes L^8_2)}$



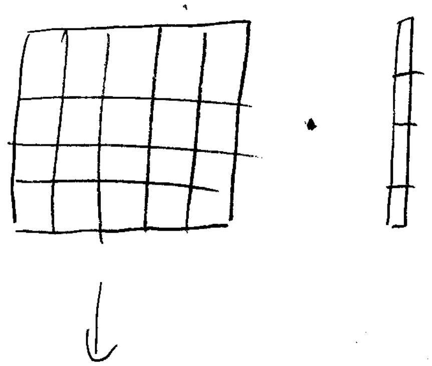
defined macro (SSE and later)

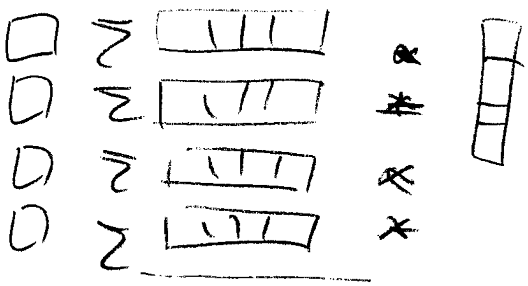
```
#define _MM_TRANSPOSE4_PS(r0,r1,r2,r3) {                \
   __m128 t0, t1, t2, t3;                               \
   t0 = _mm_shuffle_ps(r0, r1, _MM_SHUFFLE(2,0,2,0));   \
   t1 = _mm_shuffle_ps(r0, r1, _MM_SHUFFLE(3,1,3,1));   \
   t2 = _mm_shuffle_ps(r2, r3, _MM_SHUFFLE(2,0,2,0));   \
   t3 = _mm_shuffle_ps(r2, r3, _MM_SHUFFLE(3,1,3,1));   \
   r0 = _mm_shuffle_ps(t0, t2, _MM_SHUFFLE(2,0,2,0));   \
   r1 = _mm_shuffle_ps(t1, t3, _MM_SHUFFLE(2,0,2,0));   \
   r2 = _mm_shuffle_ps(t0, t2, _MM_SHUFFLE(3,1,3,1));   \
   r3 = _mm_shuffle_ps(t1, t3, _MM_SHUFFLE(3,1,3,1));   \
}
```

_MM_TRANSPOSE2_PD also exists (for doubles)

Matrix-Vector Product (SSE and later)

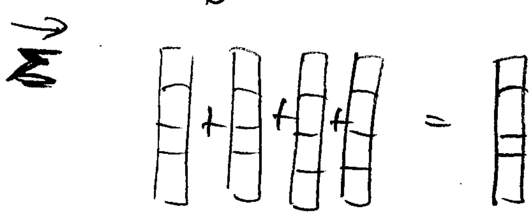
needs redrawing





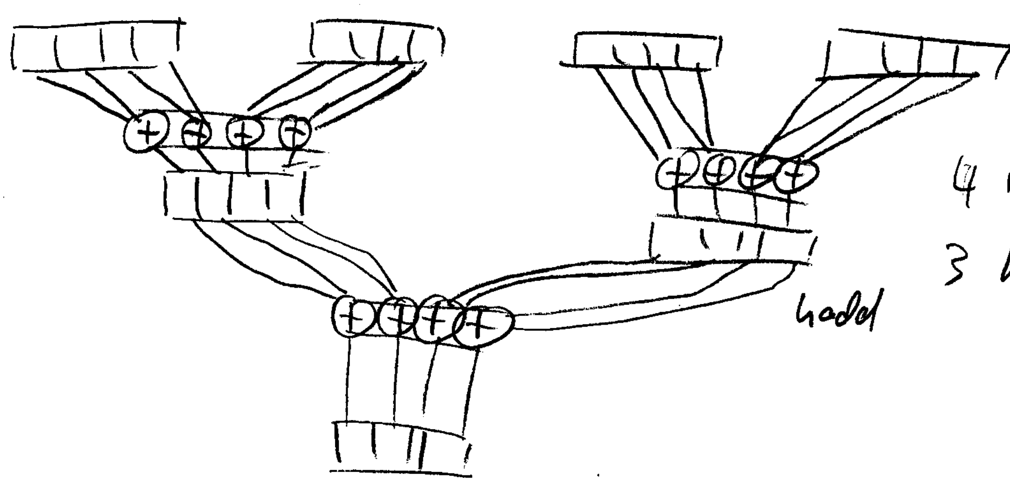
5 vector loads
4 vector mults
but how to do the sums?

transpose



3 adds, 8 shuffles

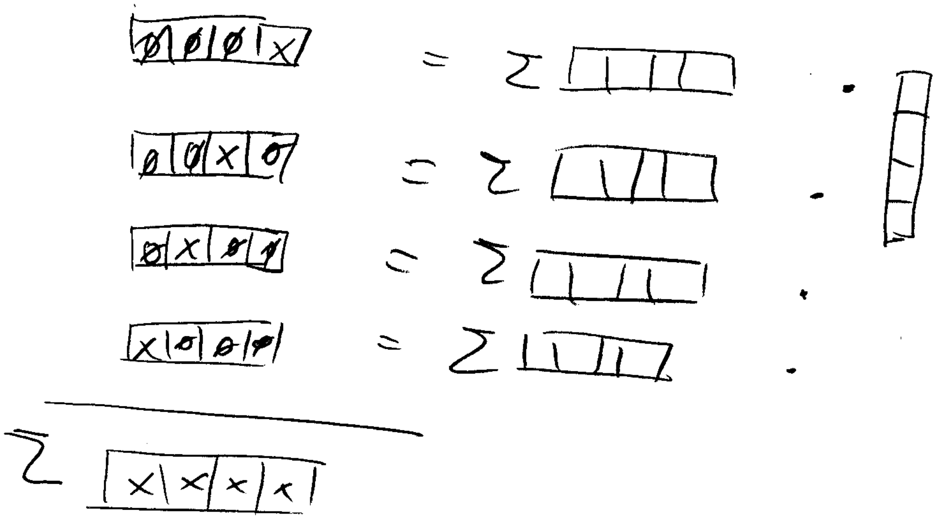
other solutions (SSE3 and higher)



hadd
4 mul
3 hadd

Matrix-Vector Product (SSE4)

needs redrawing

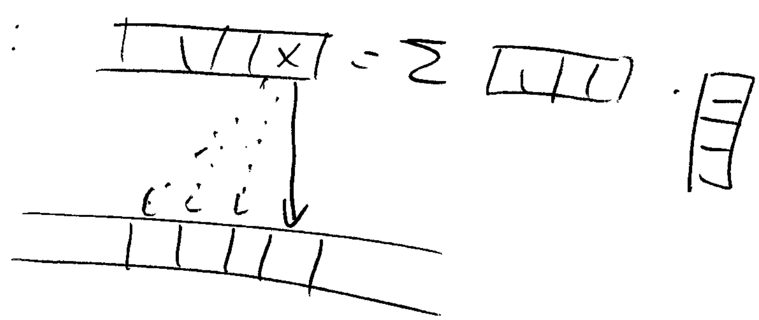


4 dot product instructions
3 add instructions

---

or 4 dot products

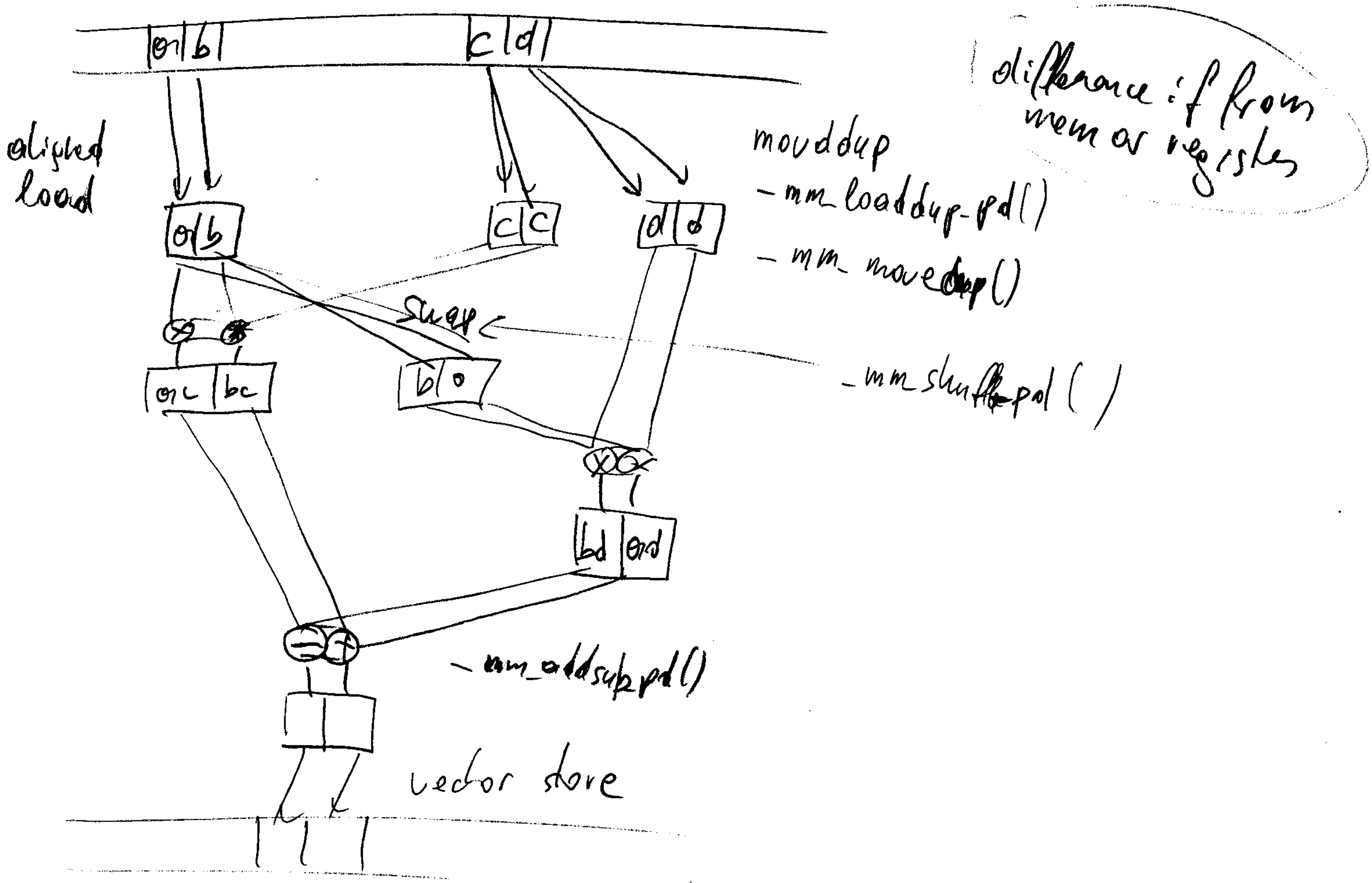
+ ~~4x~~ scalar store ops :



performance implications?

Complex Multiplication double SSE3

$$(a+ib)(c+id) = (ac-bd)+i(ad+bc)$$



SSE, SSE2: see slides