How to Write Fast Numerical Code

Spring 2012

Lecture 15

Instructor: Markus Püschel

TAs: Georg Ofenbeck & Daniele Spampinato

ETH

Eidgenössische Technische Hochschule Zürich

Swiss Federal Institute of Technology Zurich

Flynn's Taxonomy

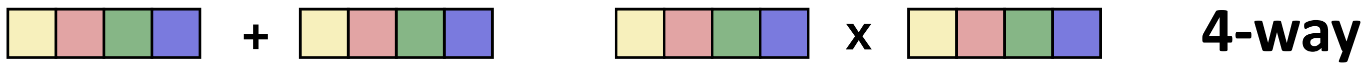
	Single instruction	**Multiple instruction**
Single data	***SISD*** Uniprocessor	***MISD***
Multiple data	***SIMD*** Vector computer Short vector extensions	***MIMD*** Multiprocessors VLIW

SIMD Extensions and SSE

- Overview: SSE family, floating point, and x87
- SSE intrinsics
- Compiler vectorization

- *This lecture and material was created together with Franz Franchetti (ECE, CMU)*

SIMD Vector Extensions

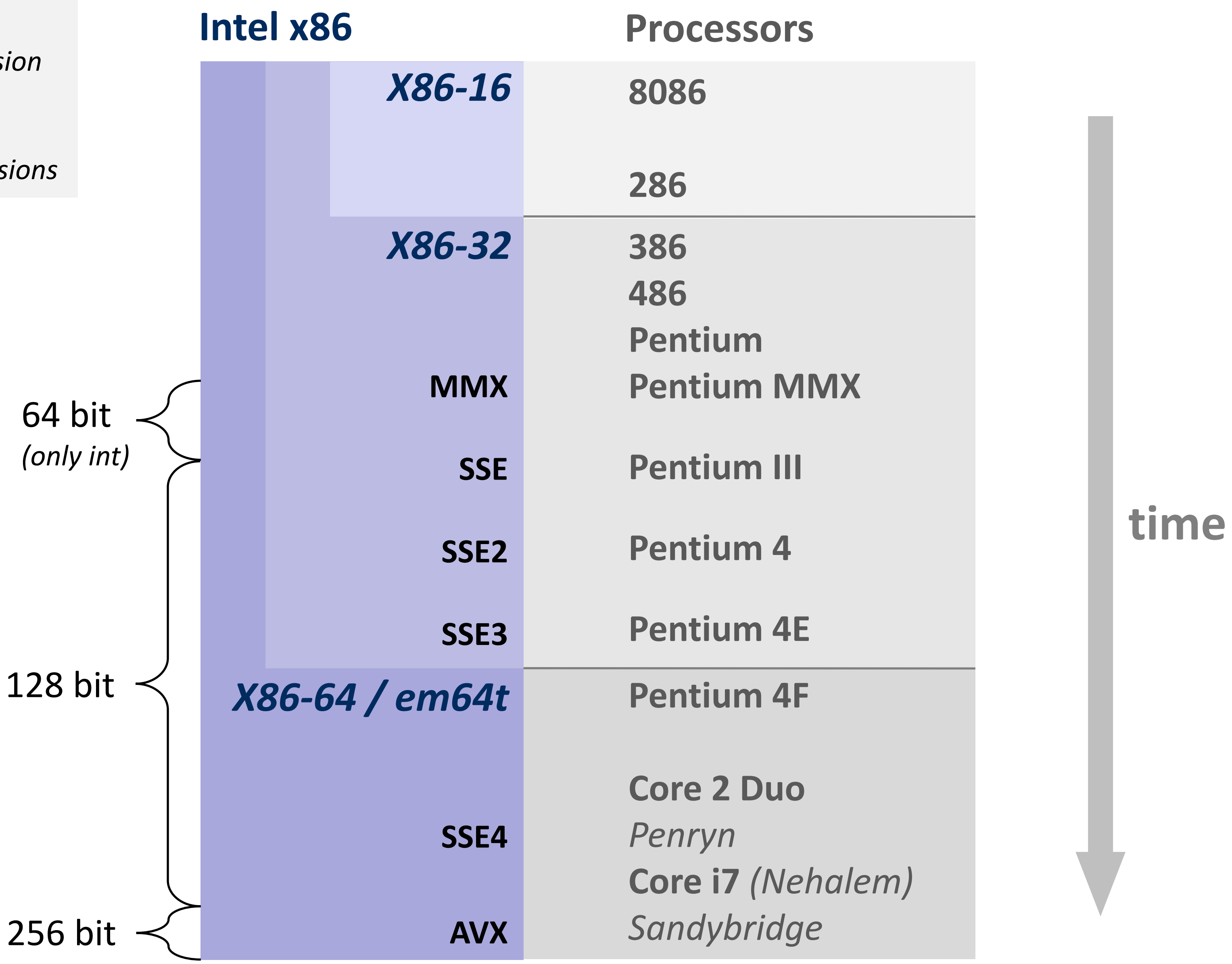


- **What is it?**
  - Extension of the ISA
  - Data types and instructions for the parallel computation on short (length 2, 4, 8, …) vectors of integers or floats
  - Names: MMX, SSE, SSE2, …
- **Why do they exist?**
  - ***Useful:*** Many applications have the necessary fine-grain parallelism
    Then: speedup by a factor close to vector length
  - ***Doable:*** Relative easy to design; chip designers have enough transistors to play with

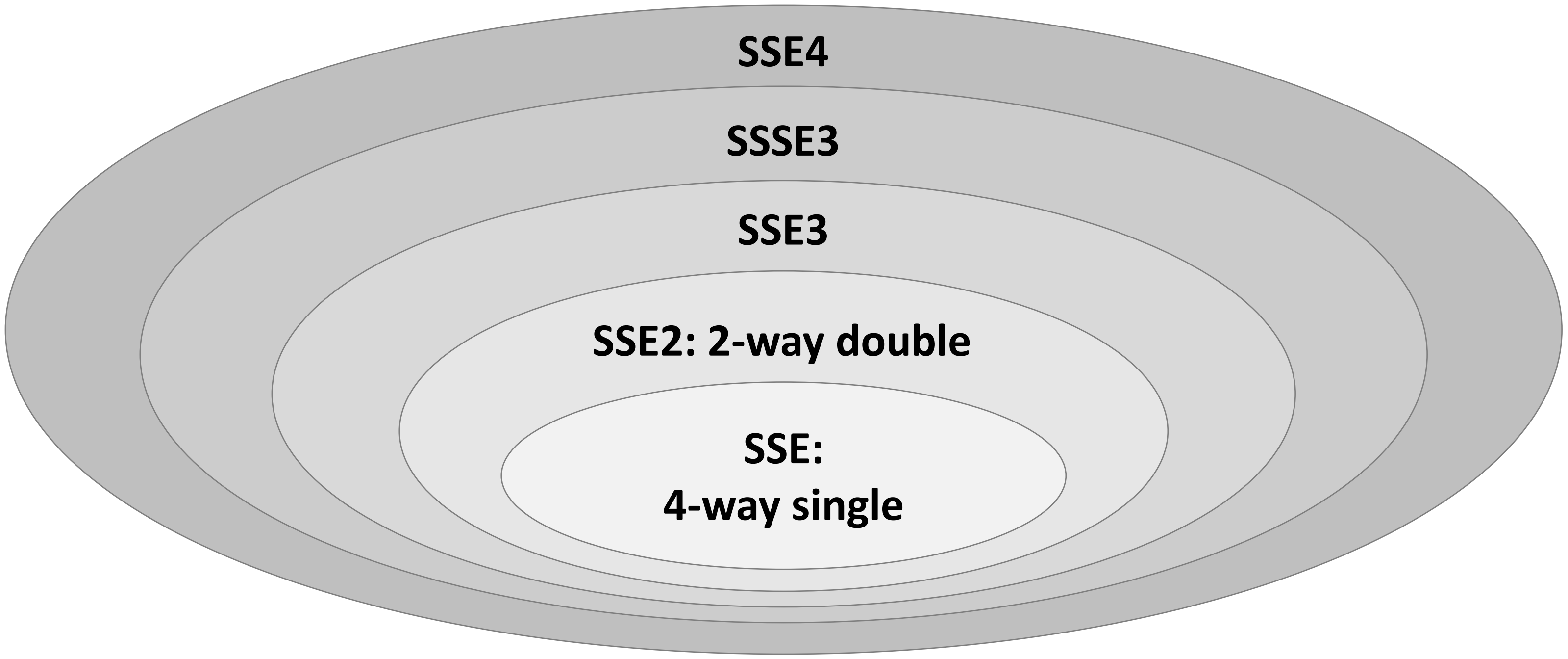
MMX:
Multimedia extension

SSE:
Streaming SIMD extension

AVX:
Advanced vector extensions



SSE Family: Floating Point



- **Not drawn to scale**
- **From SSE3: Only additional instructions**
- **Every Core 2 has SSE3**

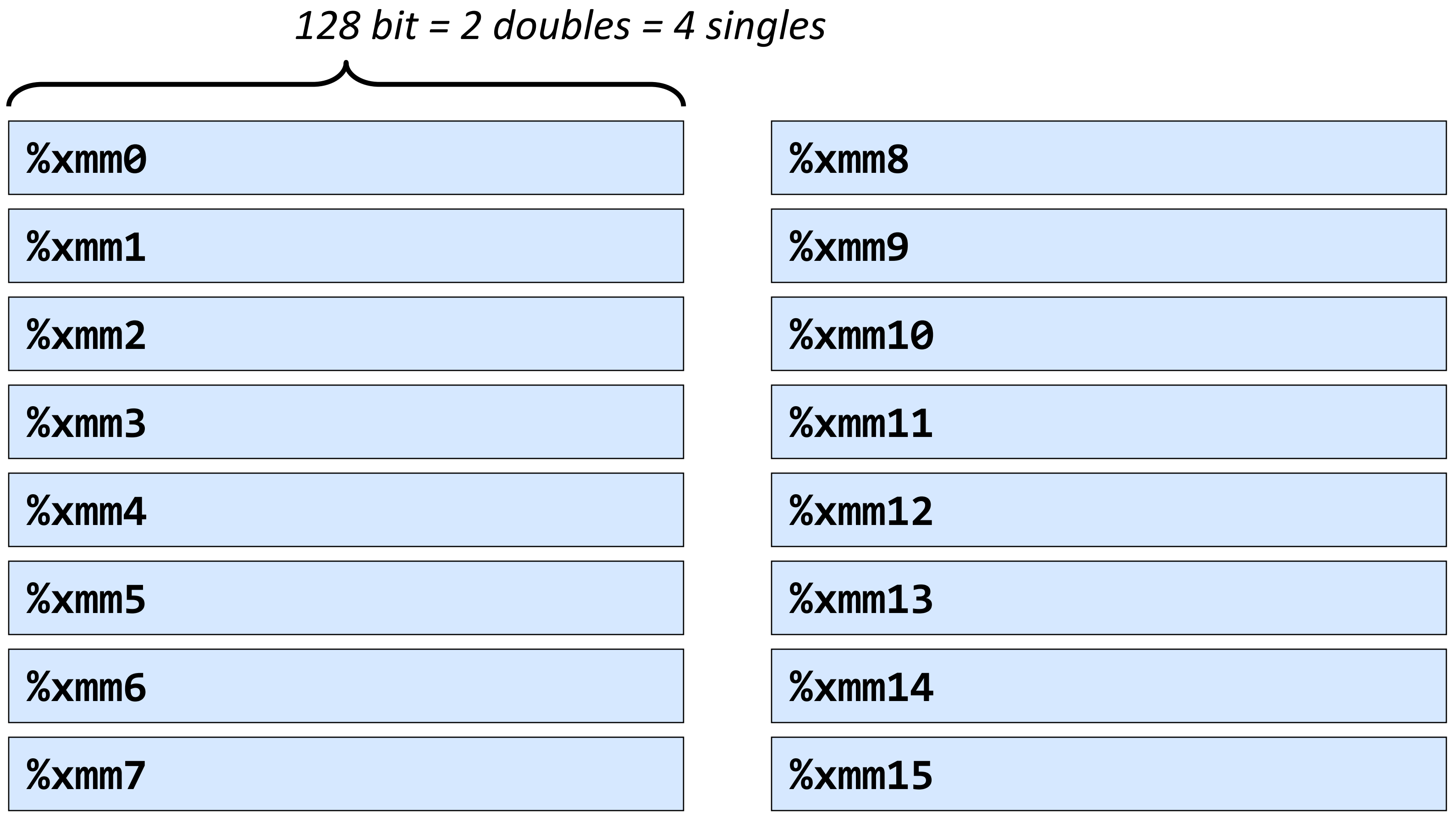
Overview Floating-Point Vector ISAs

Vendor	Name		ν-way	Precision	Introduced with
Intel	SSE		4-way	single	Pentium III
	SSE2	+	2-way	double	Pentium 4
	SSE3				Pentium 4 (Prescott)
	SSSE3				Core Duo
	SSE4				Core2 Extreme (Penryn)
	AVX		8-way	single	Core i7 (Sandybridge)
			4-way	double	
Intel	IPF		2-way	single	Itanium
Intel	LRB		16-way	single	Larrabee
			8-way	double	
AMD	3DNow!		2-way	single	K6
	Enhanced 3DNow!				K7
	3DNow! Professional	+	4-way	single	Athlon XP
	AMD64	+	2-way	double	Opteron
Motorola	AltiVec		4-way	single	MPC 7400 G4
IBM	VMX		4-way	single	PowerPC 970 G5
	SPU	+	2-way	double	Cell BE
IBM	Double FPU		2-way	double	PowerPC 440 FP2

Within an extension family, newer generations add features to older ones
Convergence: 3DNow! Professional = 3DNow! + SSE; VMX = AltiVec;

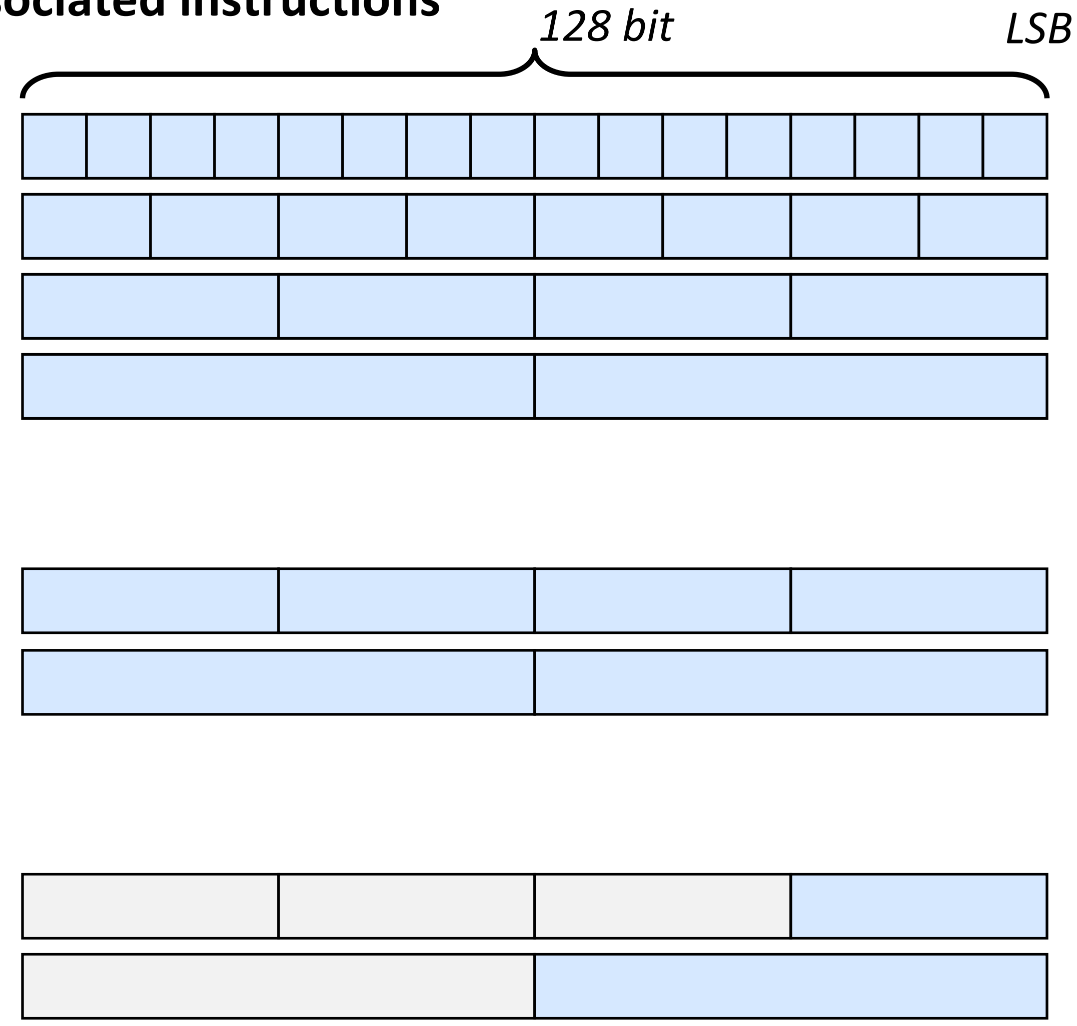
Core 2

- **Has SSE3**
- **16 SSE registers**



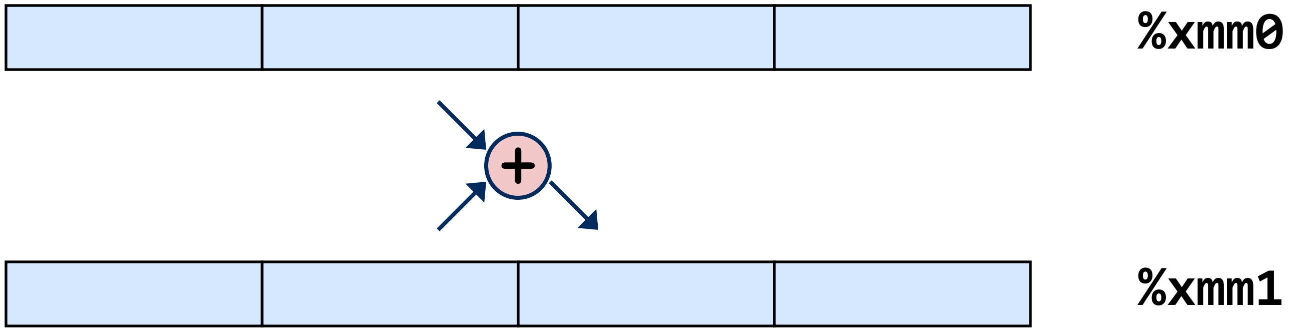
SSE3 Registers

- **Different data types and associated instructions**
- **Integer vectors:**
  - 16-way byte
  - 8-way 2 bytes
  - 4-way 4 bytes
  - 2-way 8 bytes
- **Floating point vectors:**
  - 4-way single (since SSE)
  - 2-way double (since SSE2)
- **Floating point scalars:**
  - single (since SSE)
  - double (since SSE2)

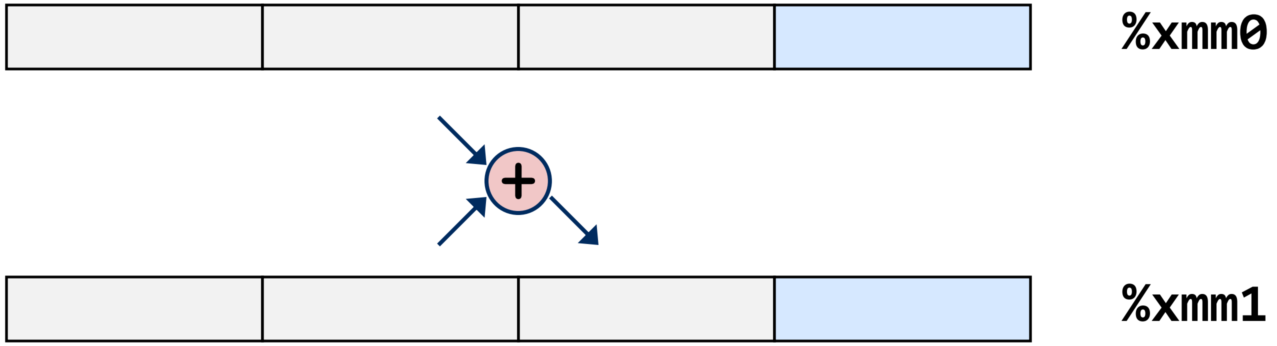


SSE3 Instructions: Examples

- Single precision *4-way vector add:* `addps %xmm0 %xmm1`



- Single precision *scalar add:* `addss %xmm0 %xmm1`



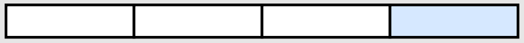
SSE3 Instruction Names

packed (vector)

single slot (scalar)

addps

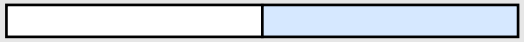
addss



single precision

addpd

addsd



double precision

Compiler will use this for floating point

- *on x86-64*
- *with proper flags if SSE/SSE2 is available*

x86-64 FP Code Example

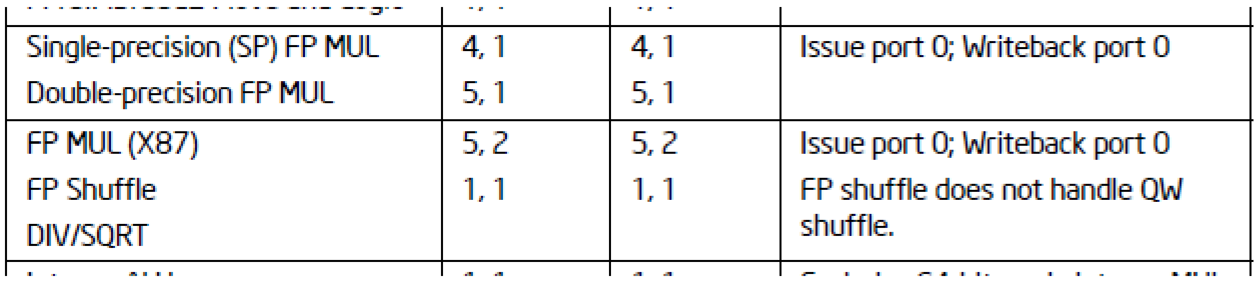
- **Inner product of two vectors**
  - Single precision arithmetic
  - Compiled: uses SSE instructions

```
float ipf (float x[],
           float y[],
           int n) {
 int i;
 float result = 0.0;

 for (i = 0; i < n; i++)
   result += x[i]*y[i];
 return result;
}
```

```
ipf:
   xorps   %xmm1, %xmm1             # result = 0.0
   xorl    %ecx, %ecx               # i = 0
   jmp     .L8                      # goto middle
.L10:                               # loop:
   movslq  %ecx,%rax                # icpy = i
   incl    %ecx                     # i++
   movss (%rsi,%rax,4), %xmm0       # t =  y[icpy]
   mulss (%rdi,%rax,4), %xmm0       # t *= x[icpy]
   addss   %xmm0, %xmm1             # result += t
.L8:                                # middle:
   cmpl    %edx, %ecx               # i:n
   jl      .L10                     # if < goto loop
   movaps  %xmm1, %xmm0             # return result
   ret
```

From Core 2 Manual

Throughput, latency



Single-precision (SP) FP MUL Double-precision FP MUL	4, 1 5, 1	4, 1 5, 1	Issue port 0; Writeback port 0
FP MUL (X87) FP Shuffle DIV/SQRT	5, 2 1, 1	5, 2 1, 1	Issue port 0; Writeback port 0 FP shuffle does not handle QW shuffle.

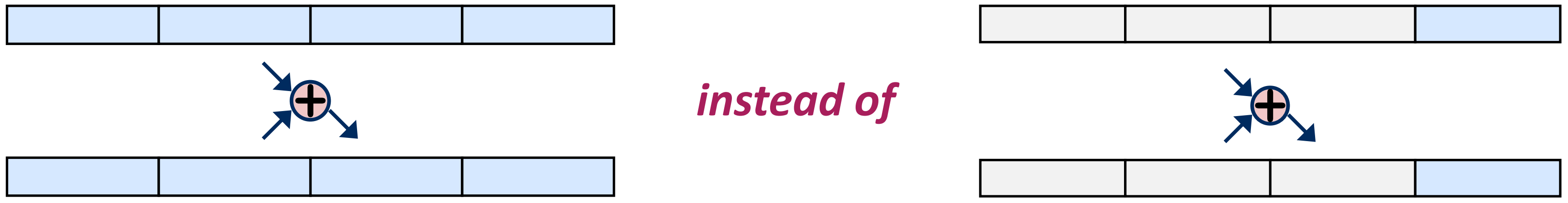
SSE based FP

x87 FP

Summary

- **On Core 2 there are two different (unvectorized) floating points**
  - x87: obsolete, is default on x86-32
  - SSE based: uses only one slot, is default on x86-64
- **SIMD vector floating point instructions**
  - 4-way single precision: since SSE
  - 2-way double precision: since SSE2
  - Since on Core 2 add and mult are fully pipelined (1 per cycle): possible gain 4x and 2x, respectively

SSE: How to Take Advantage?



- **Necessary: fine grain parallelism**
- **Options:**
  - Use vectorized libraries (easy, not always available)
  - Compiler vectorization (today)
  - Use intrinsics (today)
  - Write assembly
- **We will focus on floating point and single precision (4-way)**

SIMD Extensions and SSE

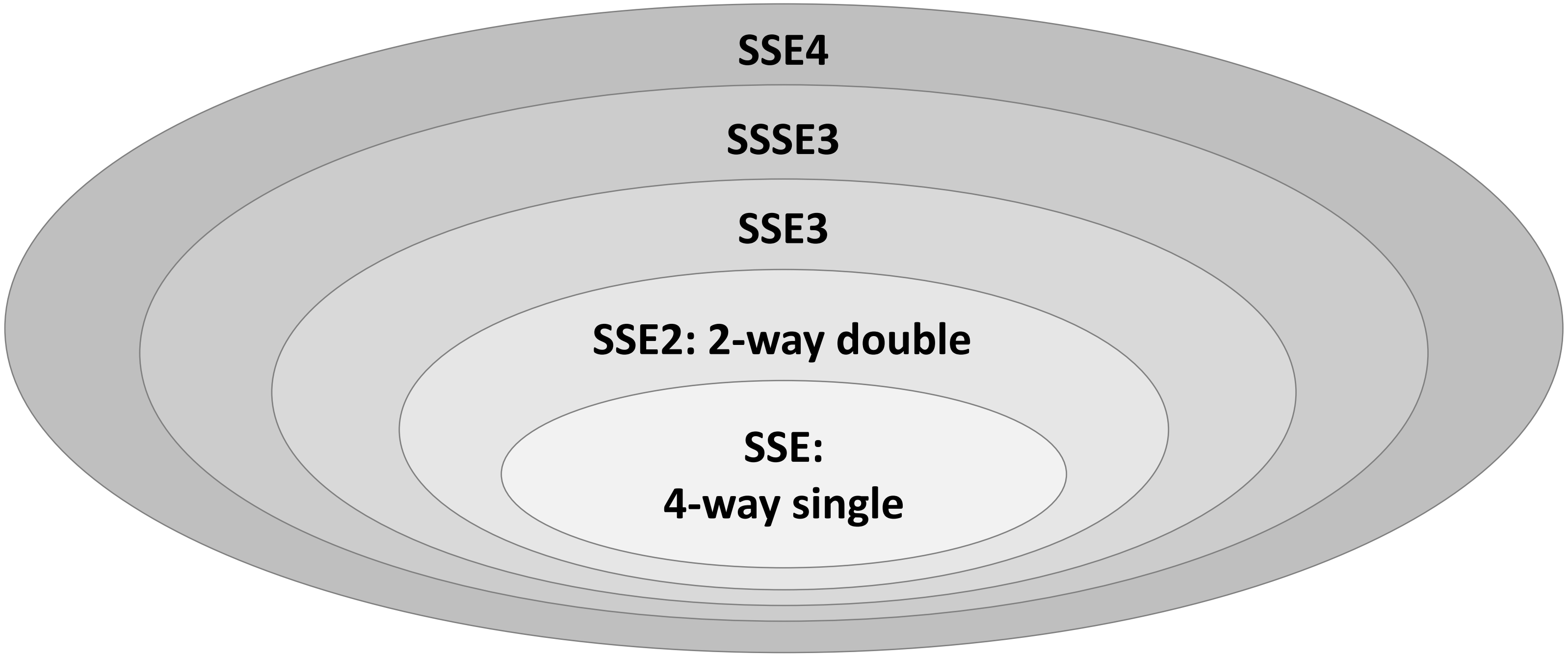
- **Overview: SSE family, floating point, and x87**
- ***SSE intrinsics***
- **Compiler vectorization**

References:

Intel icc manual (currently 12.0) → Intrinsics reference
http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/cpp/lin/index.htm

Visual Studio Manual (also: paste the intrinsic into Google)
http://msdn.microsoft.com/de-de/library/26td21ds.aspx

SSE Family: Floating Point



- Not drawn to scale
- From SSE3: Only additional instructions
- Every Core 2 has SSE3

SSE Family Intrinsics

- **Assembly coded C functions**
- **Expanded inline upon compilation: no overhead**
- **Like writing assembly inside C**
- **Floating point:**
  - Intrinsics for math functions: log, sin, …
  - Intrinsics for SSE

- **Our introduction is based on icc**
  - Most intrinsics work with gcc and Visual Studio (VS)
  - Some language extensions are icc (or even VS) specific

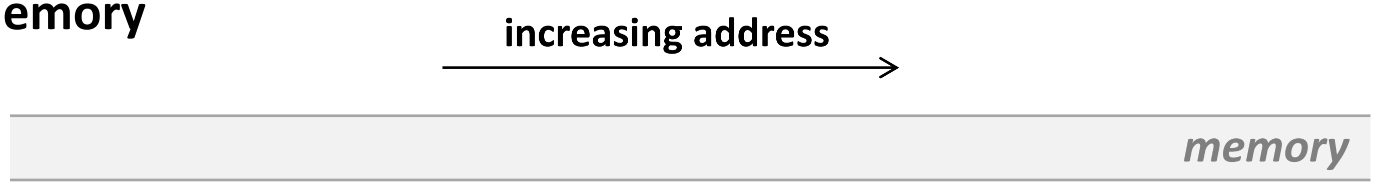
Header files

- SSE: xmmintrin.h
- SSE2: emmintrin.h
- SSE3: pmmintrin.h
- SSSE3: tmmintrin.h
- SSE4: smmintrin.h and nmmintrin.h

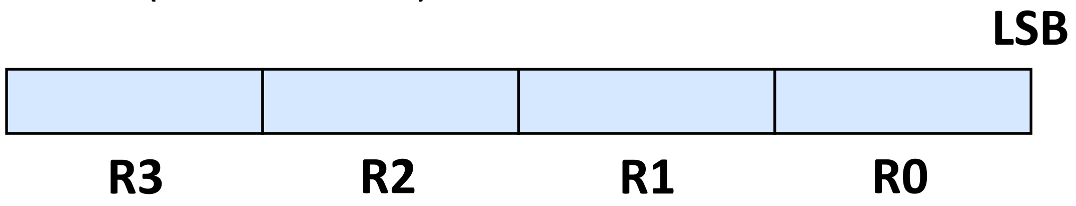
$\left.\right\}$ or ia32intrin.h

Visual Conventions We Will Use

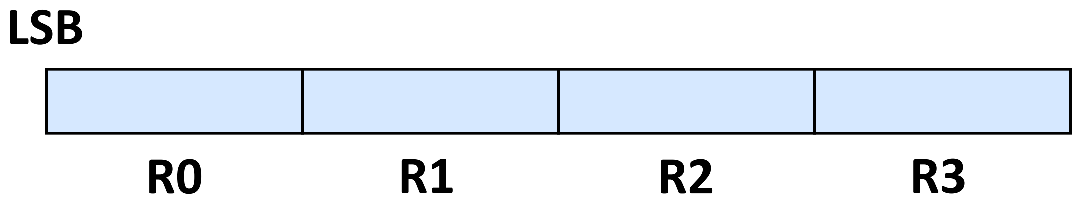
- **Memory**



- **Registers**
  - Before (and common)



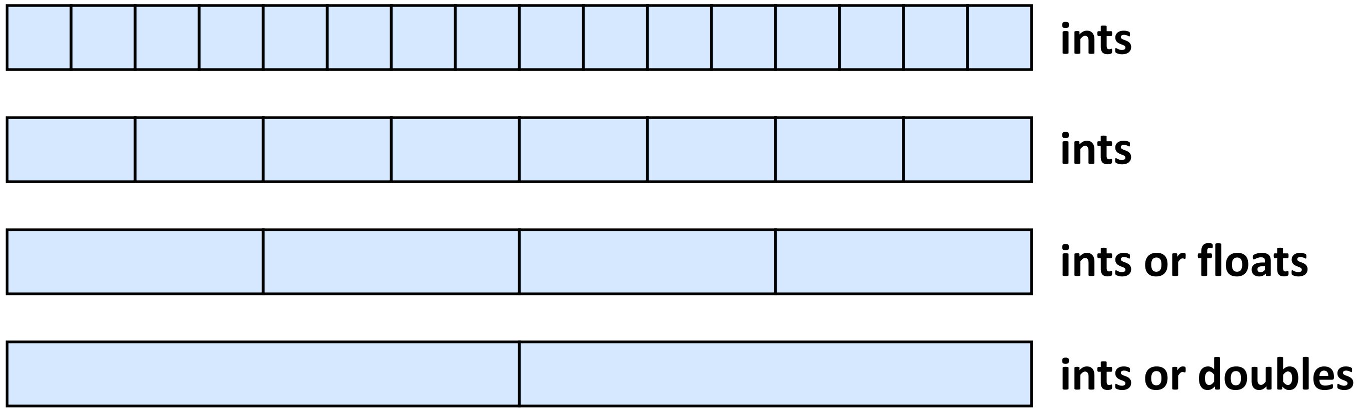
  - Now we will use



SSE Intrinsics (Focus Floating Point)

- **Data types**

```
__m128  f;    // = {float f0, f1, f2, f3}
__m128d d;    // = {double d0, d1}
__m128i i;    // 16 8-bit, 8 16-bit, 4 32-bit, or 2 64-bit ints
```

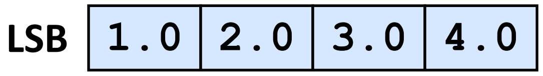


SSE Intrinsics (Focus Floating Point)

- **Instructions**
  - Naming convention: `_mm_<intrin_op>_<suffix>`
  - Example:

```
// a is 16-byte aligned
float a[4] = {1.0, 2.0, 3.0, 4.0};
__m128 t = _mm_load_ps(a);
```

p: packed
s: single precision



  - Same result as

```
__m128 t = _mm_set_ps(4.0, 3.0, 2.0, 1.0)
```

SSE Intrinsics

- **Native instructions (one-to-one with assembly)**
  ```
  _mm_load_ps()
  _mm_add_ps()
  _mm_mul_ps()
  ```
  …

- **Multi instructions (map to several assembly instructions)**
  ```
  _mm_set_ps()
  _mm_set1_ps()
  ```
  …

- **Macros and helpers**
  ```
  _MM_TRANSPOSE4_PS()
  _MM_SHUFFLE()
  ```
  …

What Are the Main Issues?

- Alignment is important (128 bit = 16 byte)
- You need to code explicit loads and stores
- Overhead through shuffles

SSE Intrinsics

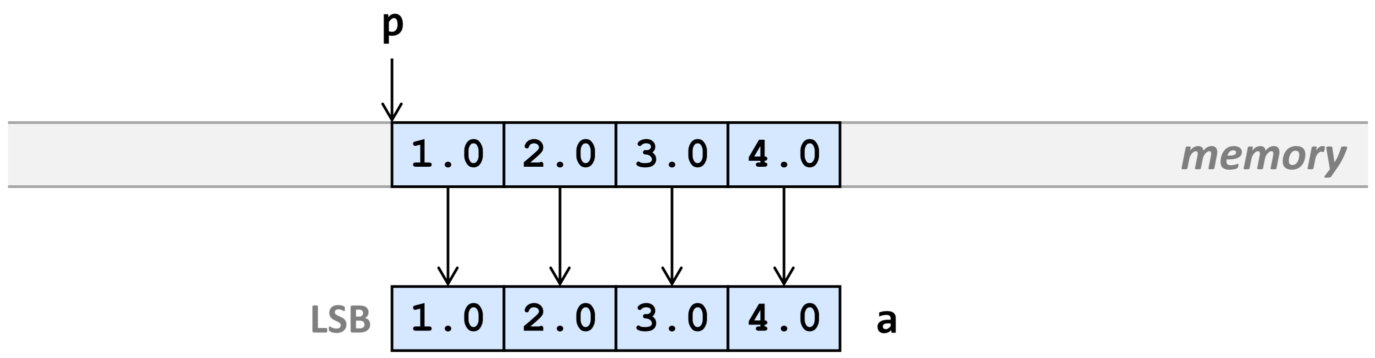
- Load and store
- Constants
- Arithmetic
- Comparison
- Conversion
- Shuffles

Loads and Stores

Intrinsic Name	Operation	Corresponding SSE Instructions
_mm_loadh_pi	Load high	MOVHPS reg, mem
_mm_loadl_pi	Load low	MOVLPS reg, mem
_mm_load_ss	Load the low value and clear the three high values	MOVSS
_mm_load1_ps	Load one value into all four words	MOVSS + Shuffling
_mm_load_ps	Load four values, address aligned	MOVAPS
_mm_loadu_ps	Load four values, address unaligned	MOVUPS
_mm_loadr_ps	Load four values in reverse	MOVAPS + Shuffling

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_set_ss	Set the low value and clear the three high values	Composite
_mm_set1_ps	Set all four words with the same value	Composite
_mm_set_ps	Set four values, address aligned	Composite
_mm_setr_ps	Set four values, in reverse order	Composite
_mm_setzero_ps	Clear all four values	Composite

Loads and Stores



```
a = _mm_load_ps(p);   // p 16-byte aligned
```

```
a = _mm_loadu_ps(p); // p not aligned
```

avoid (expensive)

→ blackboard

How to Align

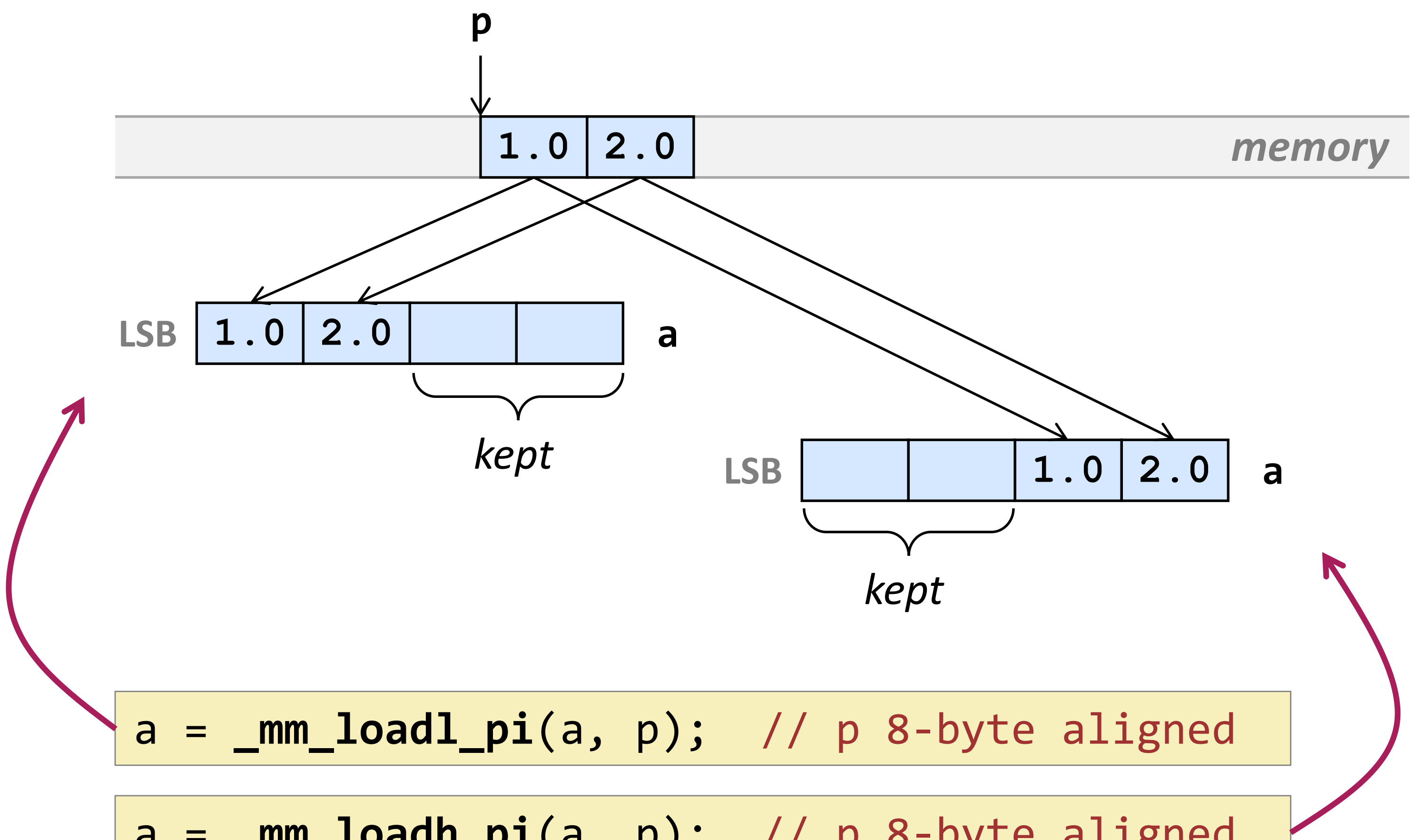
- **`__m128, __m128d, __m128i` are 16-byte aligned**

- **Arrays:**

```
__declspec(align(16)) float g[4];
```

- **Dynamic allocation**
  - _mm_malloc() and _mm_free()
  - Write your own malloc that returns 16-byte aligned addresses
  - Some malloc's already guarantee 16-byte alignment

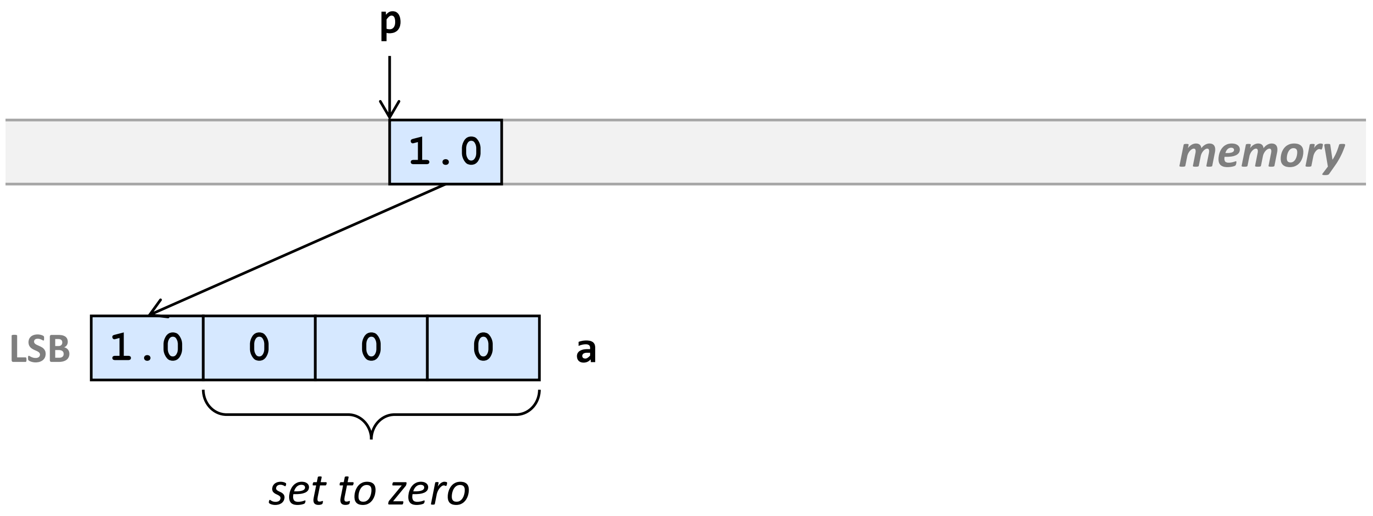
Loads and Stores



```
a = _mm_loadl_pi(a, p);  // p 8-byte aligned
```

```
a = _mm_loadh_pi(a, p);  // p 8-byte aligned
```

Loads and Stores



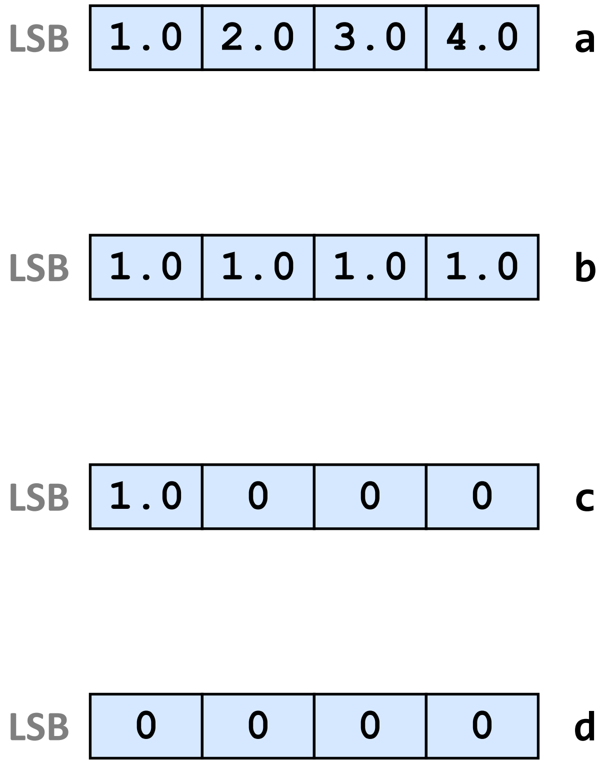
```
a = _mm_load_ss(p); // p any alignment
```

→ blackboard

Stores Analogous to Loads

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_storeh_pi	Store high	MOVHPS mem, reg
_mm_storel_pi	Store low	MOVLPS mem, reg
_mm_store_ss	Store the low value	MOVSS
_mm_store1_ps	Store the low value across all four words, address aligned	Shuffling + MOVSS
_mm_store_ps	Store four values, address aligned	MOVAPS
_mm_storeu_ps	Store four values, address unaligned	MOVUPS
_mm_storer_ps	Store four values, in reverse order	MOVAPS + Shuffling

Constants



```
a = _mm_set_ps(4.0, 3.0, 2.0, 1.0);
```

```
b = _mm_set1_ps(1.0);
```

```
c = _mm_set_ss(1.0);
```

```
d = _mm_setzero_ps();
```

→ blackboard

Arithmetic

SSE

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_add_ss	Addition	ADDSS
_mm_add_ps	Addition	ADDPS
_mm_sub_ss	Subtraction	SUBSS
_mm_sub_ps	Subtraction	SUBPS
_mm_mul_ss	Multiplication	MULSS
_mm_mul_ps	Multiplication	MULPS
_mm_div_ss	Division	DIVSS
_mm_div_ps	Division	DIVPS
_mm_sqrt_ss	Squared Root	SQRTSS
_mm_sqrt_ps	Squared Root	SQRTPS
_mm_rcp_ss	Reciprocal	RCPSS
_mm_rcp_ps	Reciprocal	RCPPS
_mm_rsqrt_ss	Reciprocal Squared Root	RSQRTSS
_mm_rsqrt_ps	Reciprocal Squared Root	RSQRTPS
_mm_min_ss	Computes Minimum	MINSS
_mm_min_ps	Computes Minimum	MINPS
_mm_max_ss	Computes Maximum	MAXSS
_mm_max_ps	Computes Maximum	MAXPS

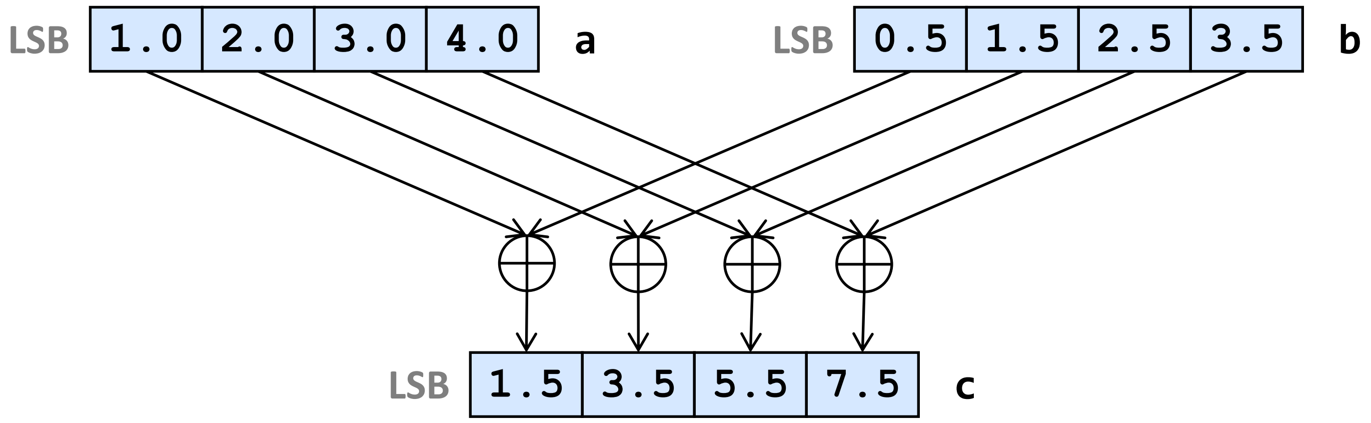
SSE3

Intrinsic Name	Operation	Corresponding SSE3 Instruction
_mm_addsub_ps	Subtract and add	ADDSUBPS
_mm_hadd_ps	Add	HADDPS
_mm_hsub_ps	Subtracts	HSUBPS

SSE4

Intrinsic	Operation	Corresponding SSE4 Instruction
_mm_dp_ps	Single precision dot product	DPPS

Arithmetic



```
c = _mm_add_ps(a, b);
```

analogous:

```
c = _mm_sub_ps(a, b);
```

```
c = _mm_mul_ps(a, b);
```

→ blackboard

Example

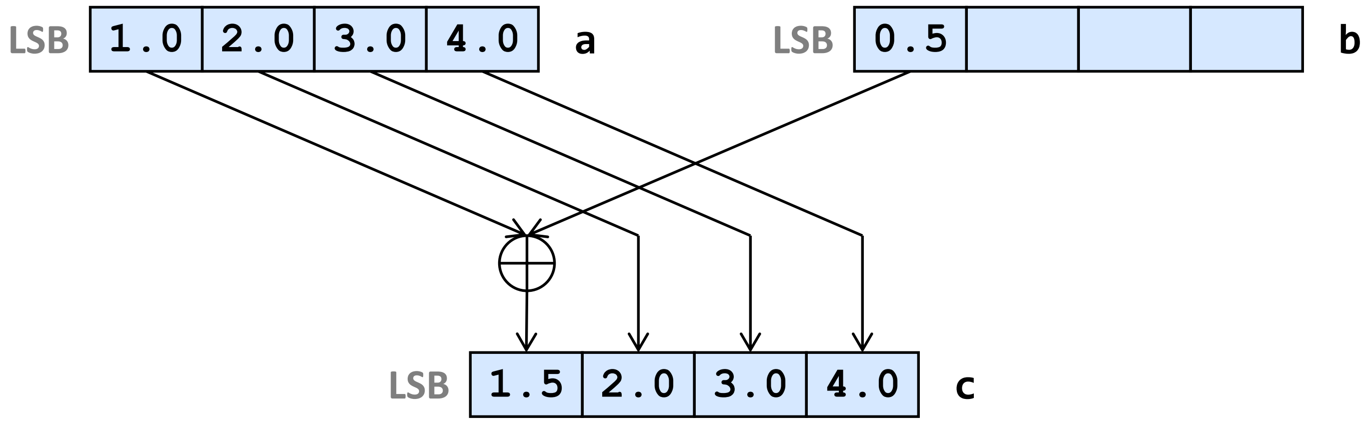
```
void addindex(float *x, int n) {
  for (int i = 0; i < n; i++)
    x[i] = x[i] + i;
}
```

```
#include <ia32intrin.h>

// n a multiple of 4, x is 16-byte aligned
void addindex_vec(float *x, int n) {
  __m128 index, x_vec;

  for (int i = 0; i < n; i+=4) {
    x_vec = _mm_load_ps(x+i);                 // load 4 floats
    index = _mm_set_ps(i+3, i+2, i+1, i);  // create vector with indexes
    x_vec = _mm_add_ps(x_vec, index);       // add the two
    _mm_store_ps(x+i, x_vec);                 // store back
  }
}
```

How does the code style differ from scalar code?

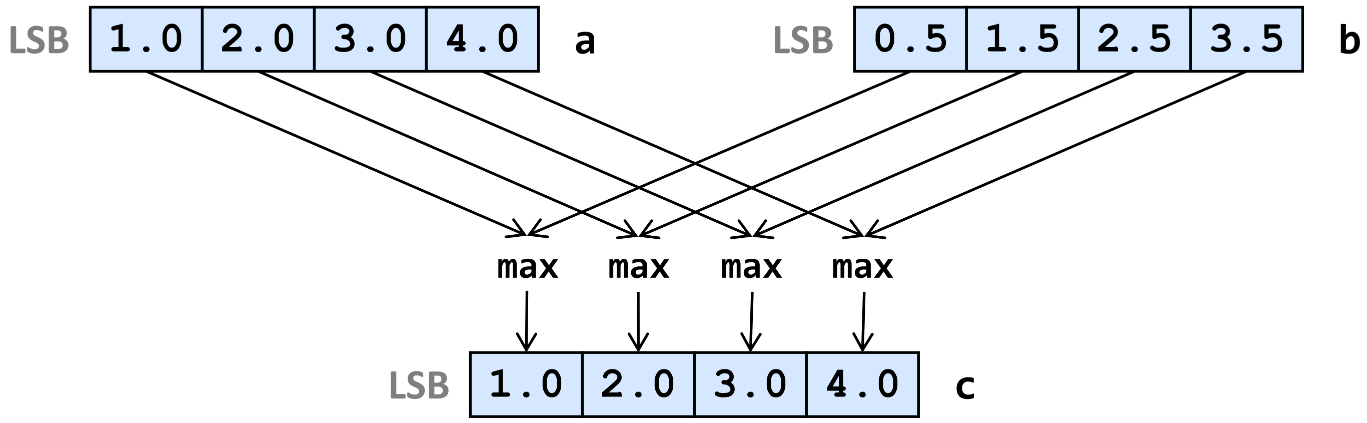
Intrinsics force scalar replacement!

Arithmetic



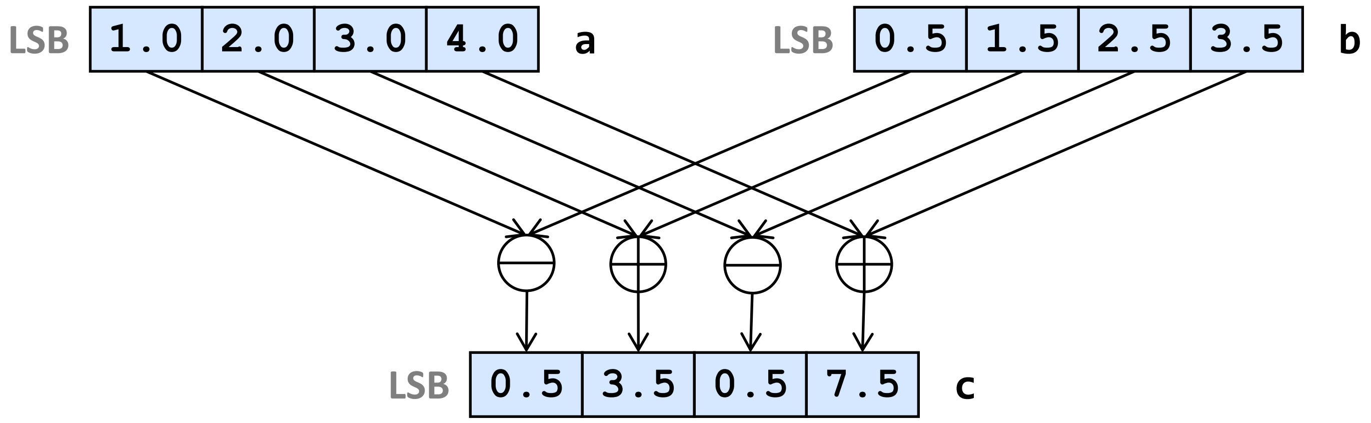
```
c = _mm_add_ss(a, b);
```

Arithmetic



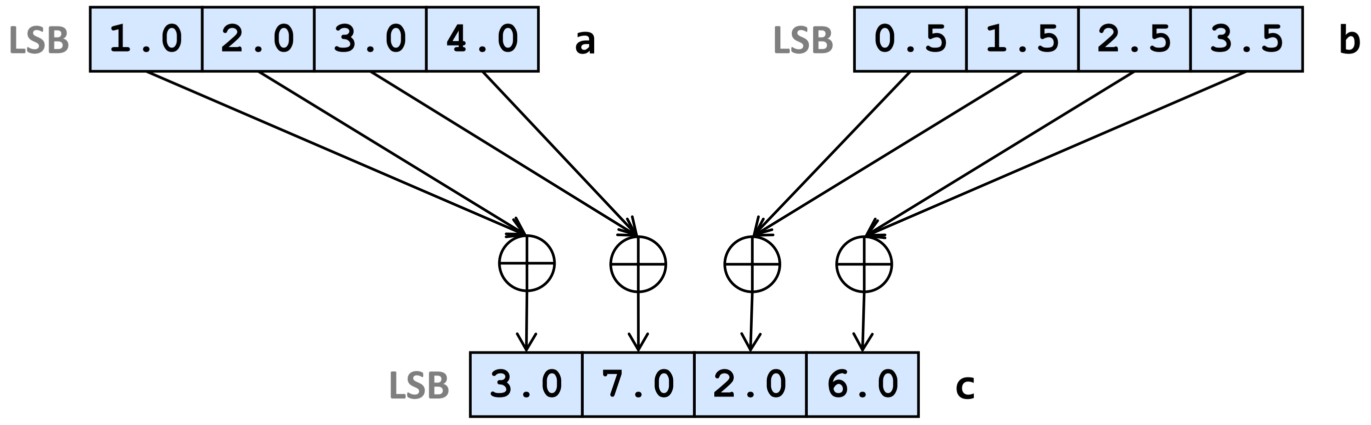
```
c = _mm_max_ps(a, b);
```

Arithmetic



```
c = _mm_addsub_ps(a, b);
```

Arithmetic



```
c = _mm_hadd_ps(a, b);
```

analogous:

```
c = _mm_hsub_ps(a, b);
```

→ blackboard

Example

```
// n is even
void lp(float *x, float *y, int n) {
  for (int i = 0; i < n/2; i++)
    y[i] = (x[2*i] + x[2*i+1])/2;
}
```

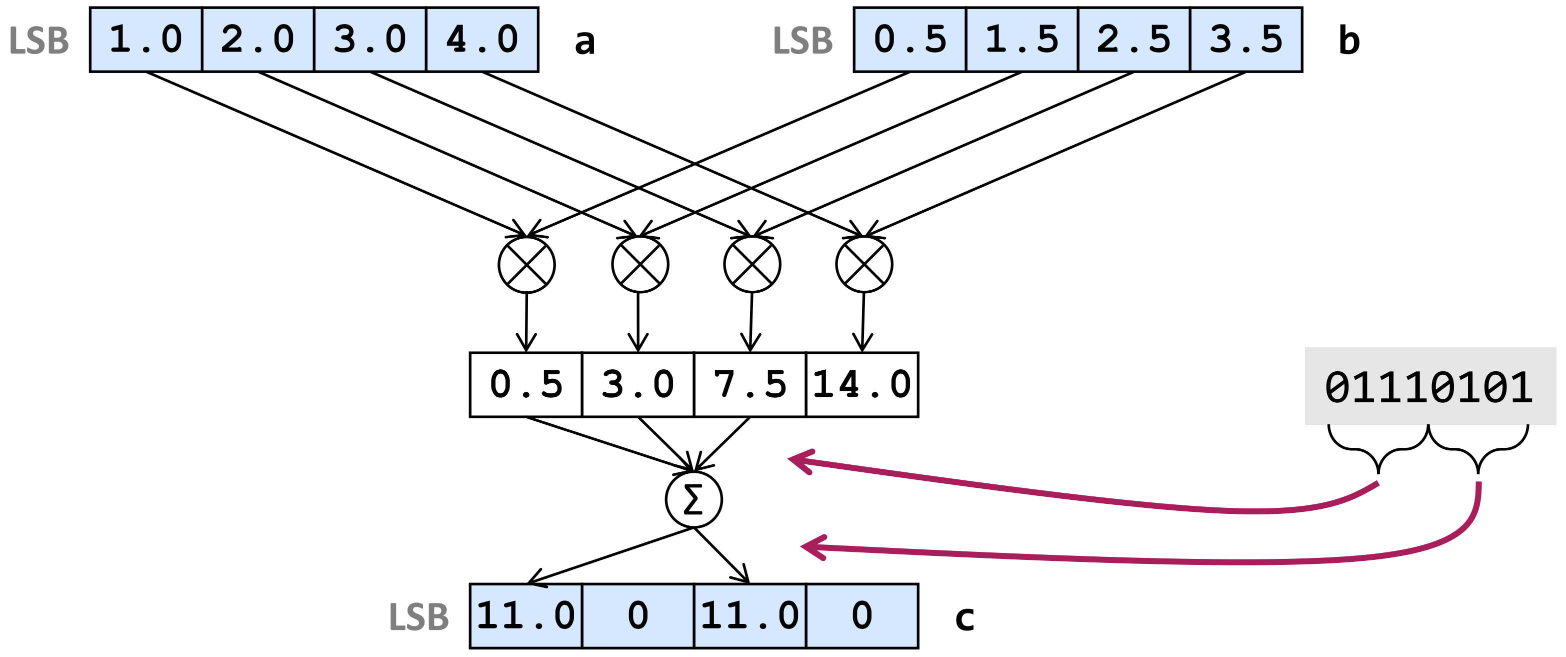
```
#include <ia32intrin.h>

// n a multiple of 8, x, y are 16-byte aligned
void lp_vec(float *x, int n) {
  __m128 half, v1, v2, avg;

  half = _mm_set1_ps(0.5);         // set vector to all 0.5
  for (int i = 0; i < n/8; i++) {
    v1  = _mm_load_ps(x+i*8);      // load first 4 floats
    v2  = _mm_load_ps(x+4+i*8);    // load next 4 floats
    avg = _mm_hadd_ps(v1, v2);     // add pairs of floats
    avg = _mm_mul_ps(avg, half);   // multiply with 0.5
    _mm_store_ps(y+i*4, avg);      // save result
  }
}
```

Arithmetic

```
__m128 _mm_dp_ps(__m128 a, __m128 b, const int mask)
```

(SSE4) Computes the pointwise product of a and b and writes a selected sum of the resulting numbers into selected elements of c; the others are set to zero. The selections are encoded in the mask.

Example: `mask = 117 = 01110101`

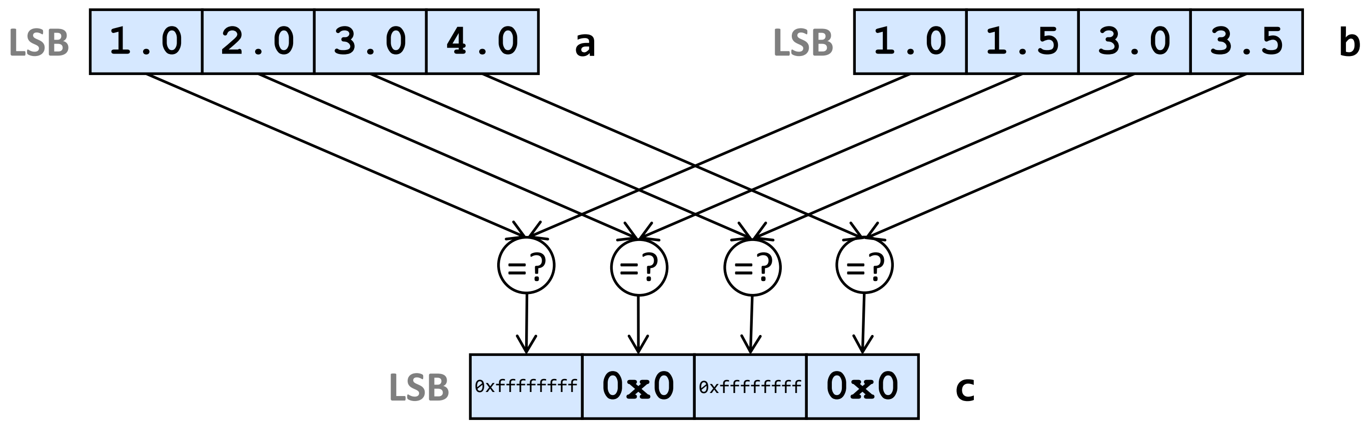


Comparisons

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_cmpeq_ss	Equal	CMPEQSS
_mm_cmpeq_ps	Equal	CMPEQPS
_mm_cmplt_ss	Less Than	CMPLTSS
_mm_cmplt_ps	Less Than	CMPLTPS
_mm_cmple_ss	Less Than or Equal	CMPLESS
_mm_cmple_ps	Less Than or Equal	CMPLEPS
_mm_cmpgt_ss	Greater Than	CMPLTSS
_mm_cmpgt_ps	Greater Than	CMPLTPS
_mm_cmpge_ss	Greater Than or Equal	CMPLESS
_mm_cmpge_ps	Greater Than or Equal	CMPLEPS
_mm_cmpneq_ss	Not Equal	CMPNEQSS
_mm_cmpneq_ps	Not Equal	CMPNEQPS
_mm_cmpnlt_ss	Not Less Than	CMPNLTSS
_mm_cmpnlt_ps	Not Less Than	CMPNLTPS
_mm_cmpnle_ss	Not Less Than or Equal	CMPNLESS
_mm_cmpnle_ps	Not Less Than or Equal	CMPNLEPS
_mm_cmpngt_ss	Not Greater Than	CMPNLTSS
_mm_cmpngt_ps	Not Greater Than	CMPNLTPS
_mm_cmpnge_ss	Not Greater Than or Equal	CMPNLESS
_mm_cmpnge_ps	Not Greater Than or Equal	CMPNLEPS

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_cmpord_ss	Ordered	CMPORDSS
_mm_cmpord_ps	Ordered	CMPORDPS
_mm_cmpunord_ss	Unordered	CMPUNORDSS
_mm_cmpunord_ps	Unordered	CMPUNORDPS
_mm_comieq_ss	Equal	COMISS
_mm_comilt_ss	Less Than	COMISS
_mm_comile_ss	Less Than or Equal	COMISS
_mm_comigt_ss	Greater Than	COMISS
_mm_comige_ss	Greater Than or Equal	COMISS
_mm_comineq_ss	Not Equal	COMISS
_mm_ucomieq_ss	Equal	UCOMISS
_mm_ucomilt_ss	Less Than	UCOMISS
_mm_ucomile_ss	Less Than or Equal	UCOMISS
_mm_ucomigt_ss	Greater Than	UCOMISS
_mm_ucomige_ss	Greater Than or Equal	UCOMISS
_mm_ucomineq_ss	Not Equal	UCOMISS

Comparisons



```
c = _mm_cmpeq_ps(a, b);
```

analogous:

```
c = _mm_cmple_ps(a, b);
```

```
c = _mm_cmplt_ps(a, b);
```

```
c = _mm_cmpge_ps(a, b);
```

etc.

Each field:
`0xffffffff` if true
`0x0` if false

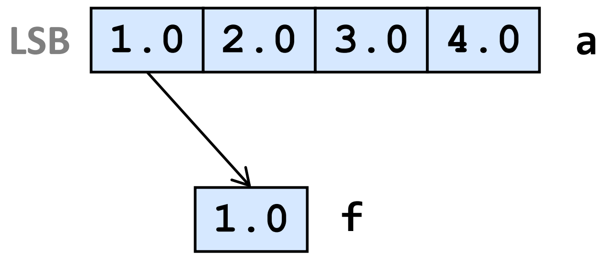
Return type: __m128

Conversion

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_cvtss_si32	Convert to 32-bit integer	CVTSS2SI
_mm_cvtss_si64*	Convert to 64-bit integer	CVTSS2SI
_mm_cvtps_pi32	Convert to two 32-bit integers	CVTPS2PI
_mm_cvttss_si32	Convert to 32-bit integer	CVTTSS2SI
_mm_cvttss_si64*	Convert to 64-bit integer	CVTTSS2SI
_mm_cvttps_pi32	Convert to two 32-bit integers	CVTTPS2PI
_mm_cvtsi32_ss	Convert from 32-bit integer	CVTSI2SS
_mm_cvtsi64_ss*	Convert from 64-bit integer	CVTSI2SS
_mm_cvtpi32_ps	Convert from two 32-bit integers	CVTTPI2PS
_mm_cvtpi16_ps	Convert from four 16-bit integers	composite
_mm_cvtpu16_ps	Convert from four 16-bit integers	composite
_mm_cvtpi8_ps	Convert from four 8-bit integers	composite
_mm_cvtpu8_ps	Convert from four 8-bit integers	composite
_mm_cvtpi32x2_ps	Convert from four 32-bit integers	composite
_mm_cvtps_pi16	Convert to four 16-bit integers	composite
_mm_cvtps_pi8	Convert to four 8-bit integers	composite
_mm_cvtss_f32	Extract	composite

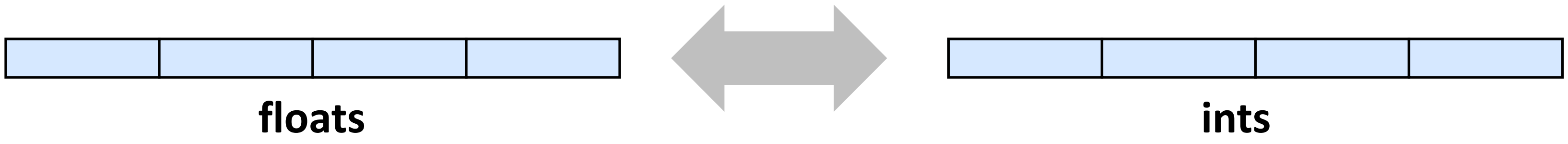
Conversion

```
float _mm_cvtss_f32(__m128 a)
```



```
float f;

f = _mm_cvtss_f32(a);
```

Cast



```
__m128i _mm_castps_si128(__m128 a)
```

```
__m128 _mm_castsi128_ps(__m128i a)
```

Reinterprets the four single precision floating point values in a as four 32-bit integers, and vice versa.

No conversion is performed. Does not map to any assembly instructions.

Makes integer shuffle instructions usable for floating point.

→ blackboard

Shuffles

SSE

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_shuffle_ps	Shuffle	SHUFPS
_mm_unpackhi_ps	Unpack High	UNPCKHPS
_mm_unpacklo_ps	Unpack Low	UNPCKLPS
_mm_move_ss	Set low word, pass in three high values	MOVSS
_mm_movehl_ps	Move High to Low	MOVHLPS
_mm_movelh_ps	Move Low to High	MOVLHPS
_mm_movemask_ps	Create four-bit mask	MOVMSKPS

SSE3

Intrinsic Name	Operation	Corresponding SSE3 Instruction
_mm_movehdup_ps	Duplicates	MOVSHDUP
_mm_moveldup_ps	Duplicates	MOVSLDUP

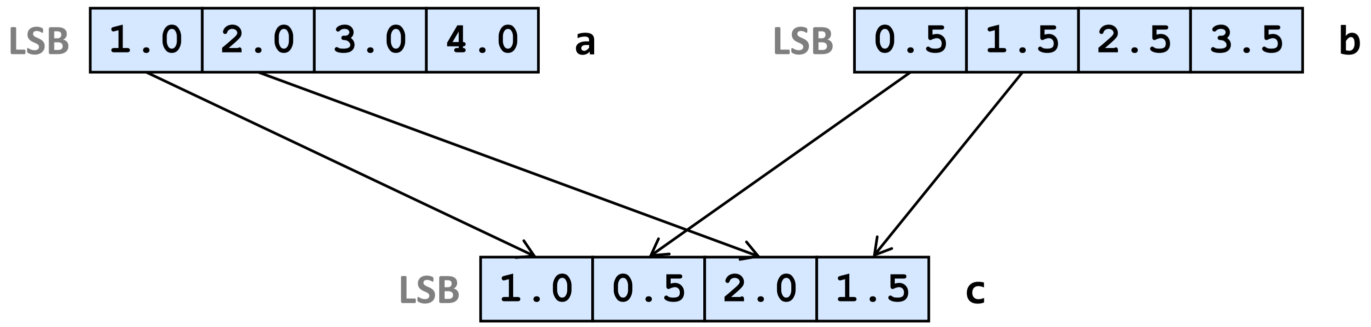
SSSE3

Intrinsic Name	Operation	Corresponding SSSE3 Instruction
_mm_shuffle_epi8	Shuffle	PSHUFB
_mm_alignr_epi8	Shift	PALIGNR

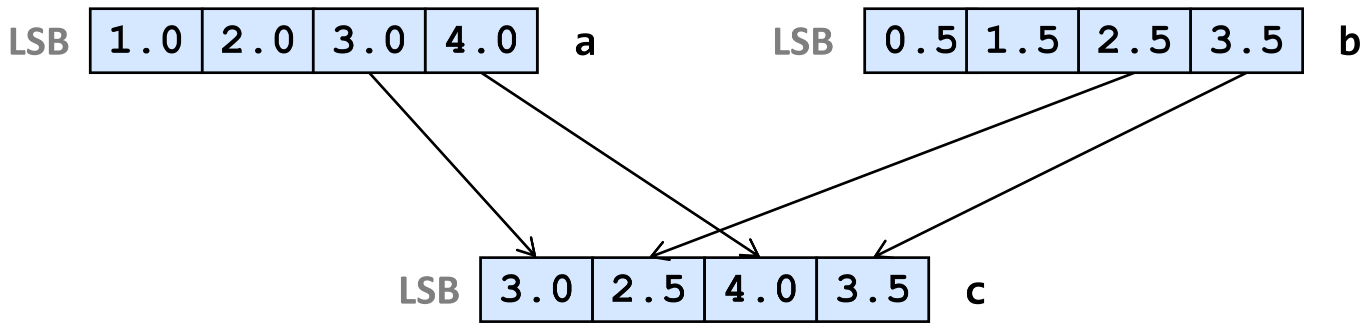
SSE4

Intrinsic Syntax	Operation	Corresponding SSE4 Instruction
__m128 _mm_blend_ps(__m128 v1, __m128 v2, const int mask)	Selects float single precision data from 2 sources using constant mask	BLENDPS
__m128 _mm_blendv_ps(__m128 v1, __m128 v2, __m128 v3)	Selects float single precision data from 2 sources using variable mask	BLENDVPS
__m128 _mm_insert_ps(__m128 dst, __m128 src, const int ndx)	Insert single precision float into packed single precision array element selected by index.	INSERTPS
int _mm_extract_ps(__m128 src, const int ndx)	Extract single precision float from packed single precision array selected by index.	EXTRACTPS

Shuffles



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



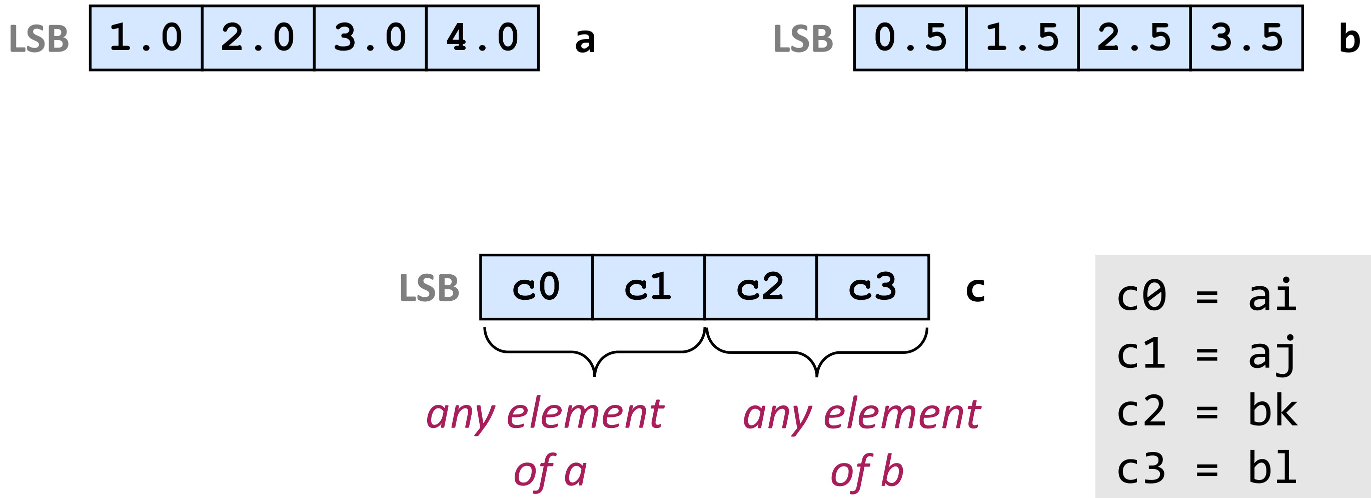
```
c = _mm_unpackhi_ps(a, b);
```

→ blackboard

Shuffles

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

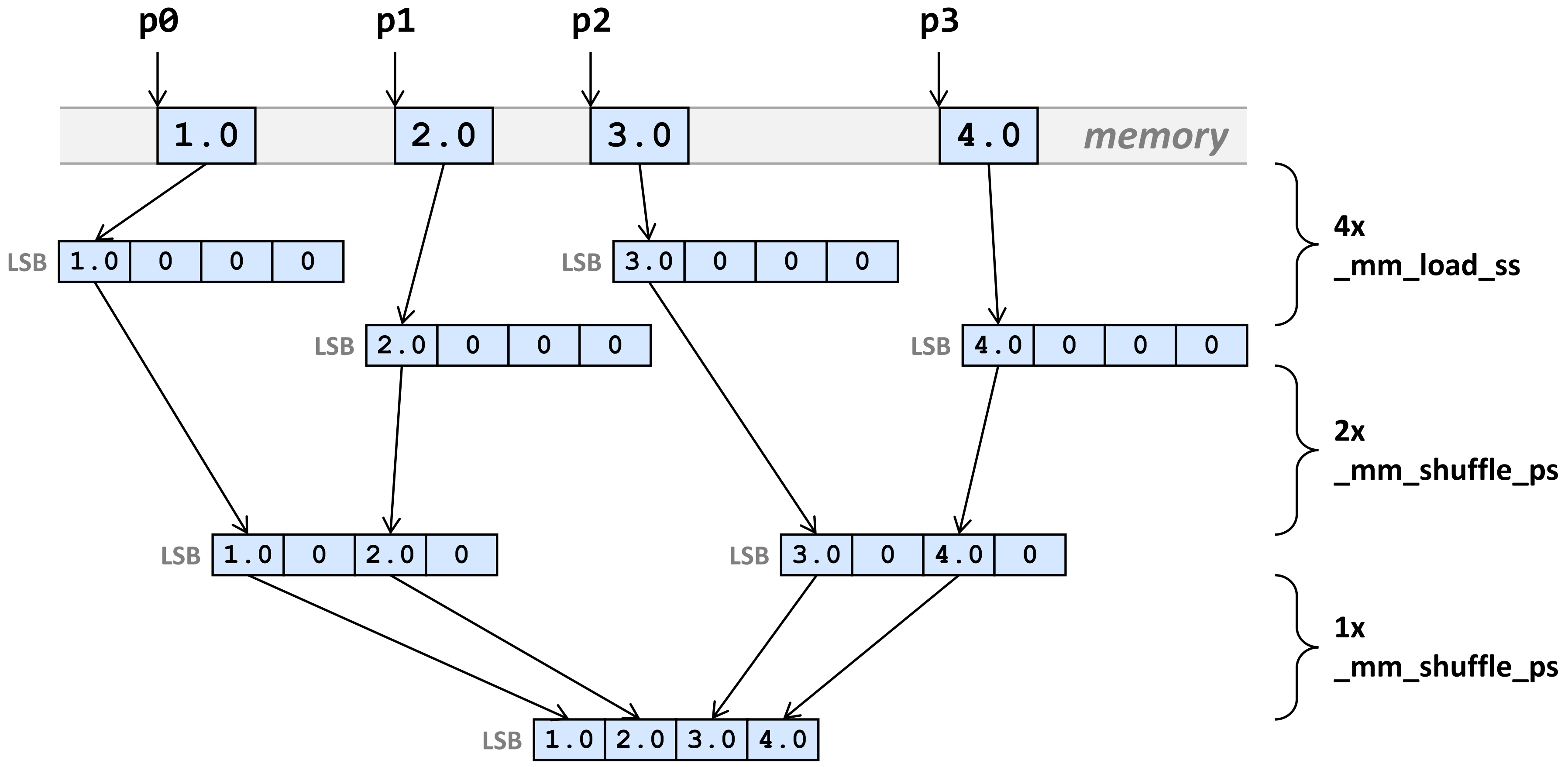
helper macro to create mask



```
c0 = ai
c1 = aj
c2 = bk
c3 = bl
i,j,k,l in {0,1,2,3}
```

→ blackboard

Example: Loading 4 Real Numbers from Arbitrary Memory Locations



7 instructions, this is the right way (before SSE4)

Code For Previous Slide

```
#include <ia32intrin.h>

__m128 LoadArbitrary(float *p0, float *p1, float *p2, float *p3) {
  __m128 a, b, c, d, e, f;

  a = _mm_load_ss(p0);
  b = _mm_load_ss(p1);
  c = _mm_load_ss(p2);
  d = _mm_load_ss(p3);
  e = _mm_shuffle_ps(a, b, _MM_SHUFFLE(1,0,2,0));    //only zeros are important
  f = _mm_shuffle_ps(c, d, _MM_SHUFFLE(1,0,2,0));    //only zeros are important
  return _mm_shuffle_ps(e, f, _MM_SHUFFLE(2,0,2,0));
}
```

Example: Loading 4 Real Numbers from Arbitrary Memory Locations (cont'd)

- **Whenever possible avoid the previous situation**
- **Restructure algorithm and use the aligned** `_mm_load_ps()`
- **Other possibility (but also yields 7 instructions and less safe)**

```
__m128 vf;

vf = _mm_set_ps(*p3, *p2, *p1, *p0);
```

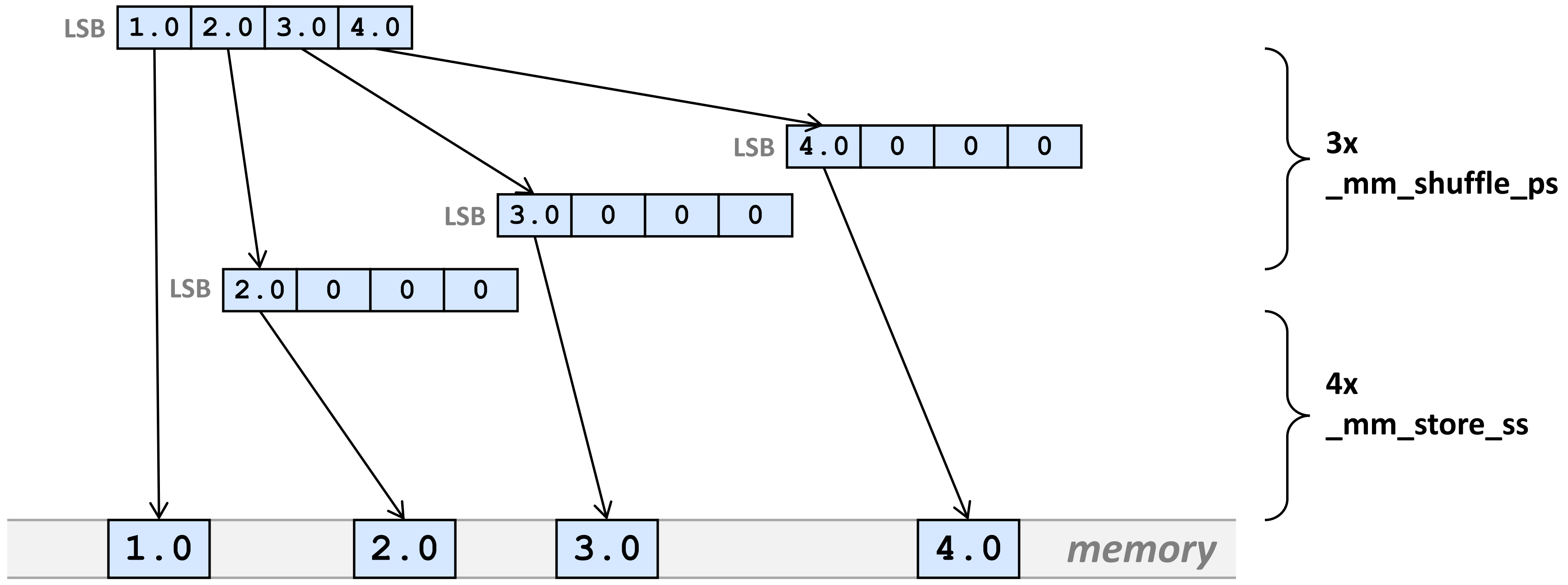
- **SSE4:** `_mm_insert_epi32` **together with** `_mm_castsi128_ps`
  - Not clear whether better

Example: Loading 4 Real Numbers from Arbitrary Memory Locations (cont'd)

- Do not do this (why?):

```
__declspec(align(16)) float g[4];
__m128 vf;

g[0] = *p0;
g[1] = *p1;
g[2] = *p2;
g[3] = *p3;
vf = _mm_load_ps(g);
```

Example: Storing 4 Real Numbers to Arbitrary Memory Locations



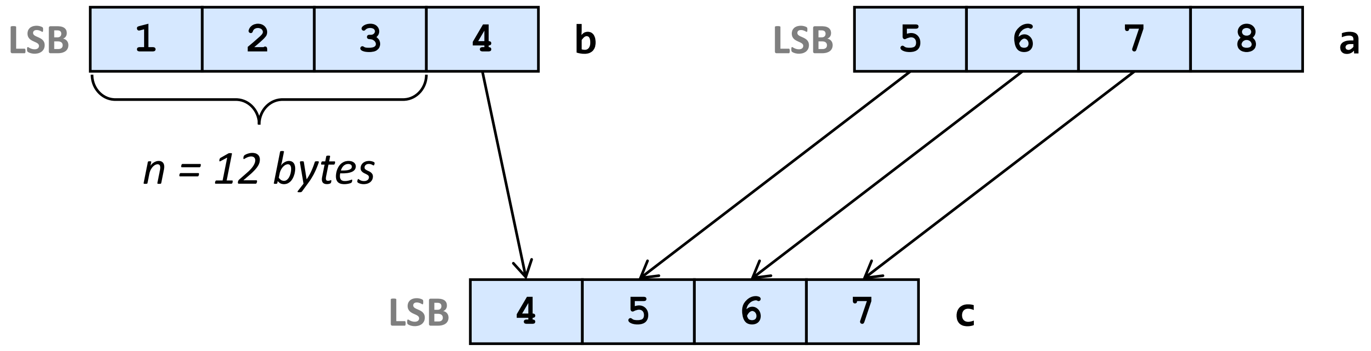
7 instructions, shorter critical path
(before SSE4)

Shuffle

```
__m128i _mm_alignr_epi8(__m128i a, __m128i b, const int n)
```

Concatenate a and b and extract byte-aligned result shifted to the right by n bytes

Example: `View __m128i as 4 32-bit ints; n = 12`



How to use this with floating point vectors?

Use with **`_mm_castsi128_ps !`**

Example

```
void shift(float *x, float *y, int n) {
  for (int i = 0; i < n-1; i++)
    y[i] = x[i+1];
  y[n-1] = 0;
}
```

Does this give a speedup?

No: bandwidth bound

```
#include <ia32intrin.h>

// n a multiple of 4, x, y are 16-byte aligned
void shift_vec(float *x, float *y, int n) {
  __m128  f;
  __m128i i1, i2, i3;

  i1 = _mm_castps_si128(_mm_load_ps(x));             // load first 4 floats and cast to int

for (int i = 0; i < n-8; i = i + 4) {
    i2 = _mm_castps_si128(_mm_load_ps(x+4+i));       // load next 4 floats and cast to int
    f  = _mm_castsi128_ps(_mm_alignr_epi8(i2,i1,4)); // shift and extract and cast back
    _mm_store_ps(y+i,f);                             // store it
    i1 = i2;                                         // make 2nd element 1st
  }

  // we are at the last 4
  i2 = _mm_castps_si128(_mm_setzero_ps());           // set the second vector to 0 and cast to int
  f  = _mm_castsi128_ps(_mm_alignr_epi8(i2,i1,4));   // shift and extract and cast back
  _mm_store_ps(y+n-4,f);                             // store it
}
```

Vectorization

=

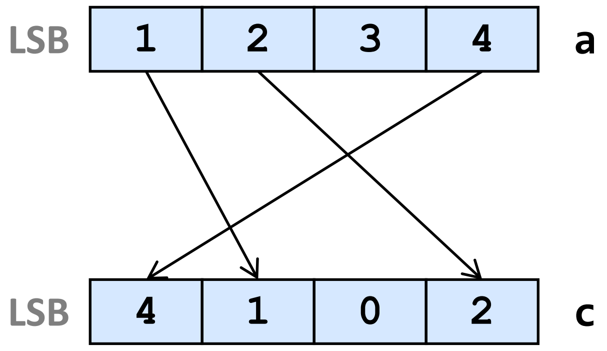
Picture: www.druckundbestell.de

Shuffle

```
__m128i _mm_shuffle_epi8(__m128i a, __m128i mask)
```

Result is filled in each position by any element of a or with 0, as specified by mask

Example: `View __m128i as 4 32-bit ints`



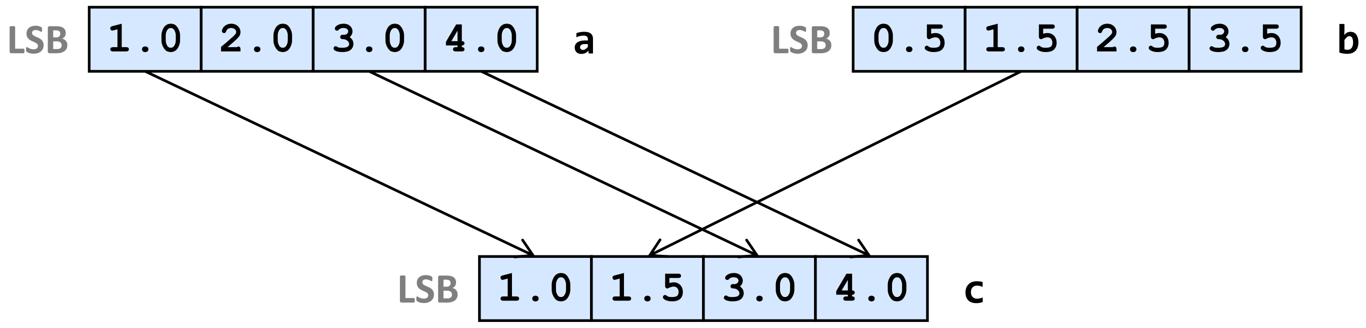
Use with `_mm_castsi128_ps` to do the same for floating point

Shuffle

```
__m128 _mm_blend_ps(__m128 a, __m128 b, const int mask)
```

(SSE4) Result is filled in each position by an element of a or b in the same position as specified by mask

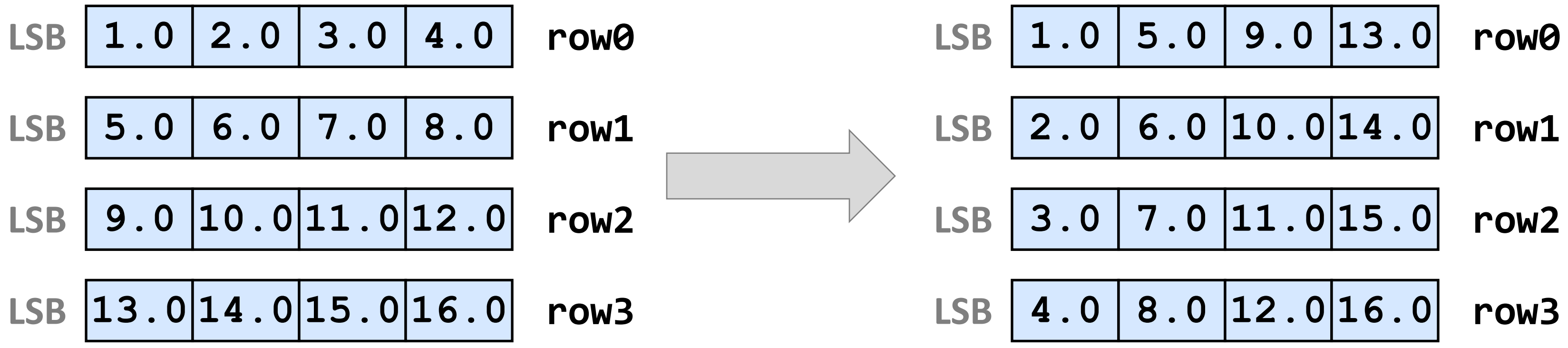
Example: `mask = 2 = 0010`



Shuffle

```
_MM_TRANSPOSE4_PS(row0, row1, row2, row3)
```

Macro for 4 x 4 matrix transposition: The arguments row0,…, row3 are __m128 values each containing a row of a 4 x 4 matrix. After execution, row0, .., row 3 contain the columns of that matrix.


LSB | 1.0 | 2.0 | 3.0 | 4.0 | row0
LSB | 5.0 | 6.0 | 7.0 | 8.0 | row1
LSB | 9.0 | 10.0 | 11.0 | 12.0 | row2
LSB | 13.0 | 14.0 | 15.0 | 16.0 | row3
LSB | 1.0 | 5.0 | 9.0 | 13.0 | row0
LSB | 2.0 | 6.0 | 10.0 | 14.0 | row1
LSB | 3.0 | 7.0 | 11.0 | 15.0 | row2
LSB | 4.0 | 8.0 | 12.0 | 16.0 | row3

In SSE: 8 shuffles (4 `_mm_unpacklo_ps`, 4 `_mm_unpackhi_ps`)

Vectorization With Intrinsics: Key Points

- **Use aligned loads and stores**
- **Minimize overhead (shuffle instructions)**
  = maximize vectorization efficiency

- ***Definition:* Vectorization efficiency**

$$\frac{\textit{Op count of scalar (unvectorized) code}}{\textit{Op count of vectorized code}} \longleftarrow$$

includes shuffles
does not include loads/stores

- ***Ideally:* Efficiency = ν for ν-way vector instructions**
  - assumes no vector instruction does more than ν scalar ops
  - assumes every vector instruction has the same cost

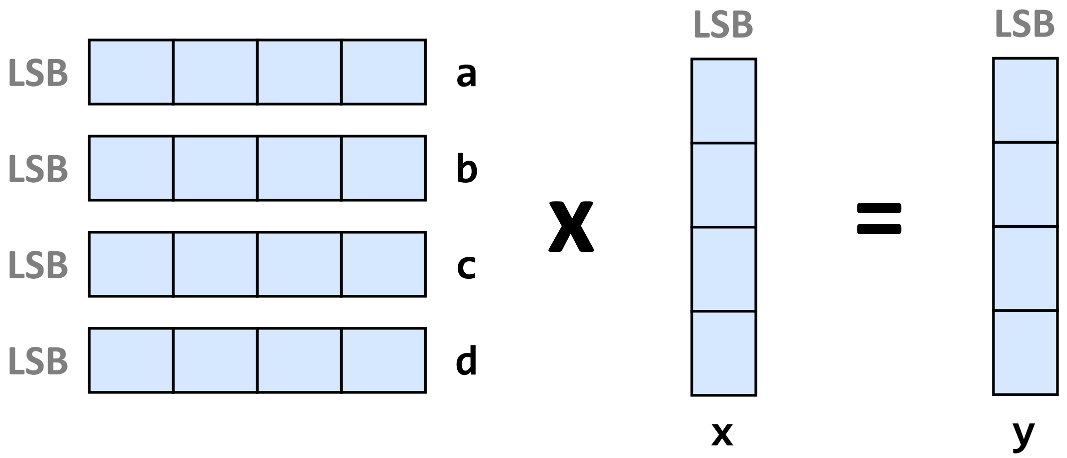
Vectorization Efficiency: Example I

```
// n is even
void lp(float *x, float *y, int n) {
  for (int i = 0; i < n/2; i++)
    y[i] = (x[2*i] + x[2*i+1])/2;
}
```

```
#include <ia32intrin.h>

// n a multiple of 8, x, y are 16-byte aligned
void lp_vec(float *x, int n) {
  __m128 half, v1, v2, avg;

  half = _mm_set1_ps(0.5);          // set vector to all 0.5
  for (int i = 0; i < n/8; i++) {
    v1  = _mm_load_ps(x+i*8);       // load first 4 floats
    v2  = _mm_load_ps(x+4+i*8);     // load next 4 floats
    avg = _mm_hadd_ps(v1, v2);      // add pairs of floats
    avg = _mm_mul_ps(avg, half);    // multiply with 0.5
    _mm_store_ps(y+i*4, avg);       // save result
  }
}
```

Vectorization Efficiency: Example 2

- 4 x 4 matrix-vector multiplication
- *Blackboard*



SIMD Extensions and SSE

- **Overview: SSE family, floating point, and x87**
- **SSE intrinsics**
- ***Compiler vectorization***

References:

Intel icc manual (currently 12.0) → Creating parallel applications → Automatic vectorization

http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/cpp/lin/index.htm

Compiler Vectorization

- Compiler flags
- Aliasing
- Proper code style
- Alignment

Compiler Flags (icc 12.0)

Linux* OS and Mac OS* X	Windows* OS	Description
-vec -no-vec	/Qvec /Qvec-	Enables or disables vectorization and transformations enabled for vectorization. Vectorization is enabled by default. To disable, use -no-vec (Linux* and MacOS* X) or /Qvec- (Windows*) option. Supported on IA-32 and Intel® 64 architectures only.
-vec-report	/Qvec-report	Controls the diagnostic messages from the vectorizer. See Vectorization Report.
-simd -no-simd	/Qsimd /Qsimd-	Controls user-mandated (SIMD) vectorization. User-mandated (SIMD) vectorization is enabled by default. Use the -no-simd (Linux* or MacOS* X) or /Qsimd- (Windows*) option to disable SIMD transformations for vectorization.

Architecture flags:

Linux: `-x`*`Host`* ⊃ `-m`*`Host`*

Windows: `/Qx`*`Host`* ⊃ `/Qarch:`*`Host`*

`Host` **in {SSE2, SSE3, SSSE3, SSE4.1, SSE4.2}**

Default: `-mSSE2, /Qarch:SSE2`

How Do I Know the Compiler Vectorized?

- `vec-report` **(previous slide)**
- **Look at assembly:** `mulps, addps, xxxps`
- **Generate assembly with source code annotation:**
  - Visual Studio + icc: /Fas
  - icc on Linux/Mac: -S

Example

```
void myadd(float *a, float *b, const int n) {
  for (int i = 0; i< n; i++)
    a[i] = a[i] + b[i];
}
```

unvectorized: /Qvec-

```
<more>
;;;    a[i] = a[i] + b[i];
movss      xmm0, DWORD PTR [rcx+rax*4]
addss      xmm0, DWORD PTR [rdx+rax*4]
movss      DWORD PTR [rcx+rax*4], xmm0
<more>
```

vectorized:

```
<more>
;;;    a[i] = a[i] + b[i];
movss      xmm0, DWORD PTR [rcx+r11*4]
addss      xmm0, DWORD PTR [rdx+r11*4]
movss      DWORD PTR [rcx+r11*4], xmm0
…
movups     xmm0, XMMWORD PTR [rdx+r10*4]
movups     xmm1, XMMWORD PTR [16+rdx+r10*4]
addps      xmm0, XMMWORD PTR [rcx+r10*4]
addps      xmm1, XMMWORD PTR [16+rcx+r10*4]
movaps     XMMWORD PTR [rcx+r10*4], xmm0
movaps     XMMWORD PTR [16+rcx+r10*4], xmm1
<more>
```

why this?

why everything twice?

why movups and movaps?

unaligned aligned

Aliasing

```
for (i = 0; i < n; i++)
  a[i] = a[i] + b[i];
```

Cannot be vectorized in a straightforward way due to potential aliasing.

However, in this case compiler can insert runtime check:

```
if (a + n < b || b + n < a)
  /* vectorized loop */
  ...
else
  /* serial loop */
  ...
```

Removing Aliasing

- **Globally with compiler flag:**
  - `-fno-alias, /Oa`
  - `-fargument-noalias, /Qalias-args-` (function arguments only)
- **For one loop: pragma**

```
void add(float *a, float *b, int n) {
  #pragma ivdep
  for (i = 0; i < n; i++)
    a[i] = a[i] + b[i];
}
```

- **For specific arrays: restrict (needs compiler flag `–restrict, /Qrestrict`)**

```
void add(float *restrict a, float *restrict b, int n) {
for (i = 0; i < n; i++)
    a[i] = a[i] + b[i];
}
```

Proper Code Style

- **Use countable loops = number of iterations known at runtime**
  - *Number of iterations is a:*
    constant
    loop invariant term
    linear function of outermost loop indices

- **Countable or not?**

```
for (i = 0; i < n; i++)
  a[i] = a[i] + b[i];
```

```
void vsum(float *a, float *b, float *c) {
  int i = 0;

  while (a[i] > 0.0) {
    a[i] = b[i] * c[i];
    i++;
  }
}
```

Proper Code Style

- **Use arrays, structs of arrays, not arrays of structs**
- **Ideally: unit stride access in innermost loop**

```
void mmm1(float *a, float *b, float *c) {
  int N = 100;
  int i, j, k;

  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
```

```
void mmm2(float *a, float *b, float *c) {
  int N = 100;
  int i, j, k;

  for (i = 0; i < N; i++)
    for (k = 0; k < N; k++)
      for (j = 0; j < N; j++)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
```

Alignment

```
float x[1024];
int i;

for (i = 0; i < 1024; i++)
   x[i] = 1;
```

Cannot be vectorized in a straightforward way since x may not be aligned

However, the compiler can peel the loop to extract aligned part:

```
float x[1024];
int i;

peel = x & 0x0f; /* x mod 16 */
if (peel != 0) {
  peel = 16 - peel;
  /* initial segment */
  for (i = 0; i < peel; i++)
    x[i] = 1;
}
/* 16-byte aligned access */
for (i = peel; i < 1024; i++)
  x[i] = 1;
```

Ensuring Alignment

- **Align arrays to 16-byte boundaries (see earlier discussion)**
- **If compiler cannot analyze:**
  - Use pragma for loops

```
float x[1024];
int i;

#pragma vector aligned
for (i = 0; i < 1024; i++)
   x[i] = 1;
```

  - For specific arrays:
    `__assume_aligned(a, 16);`

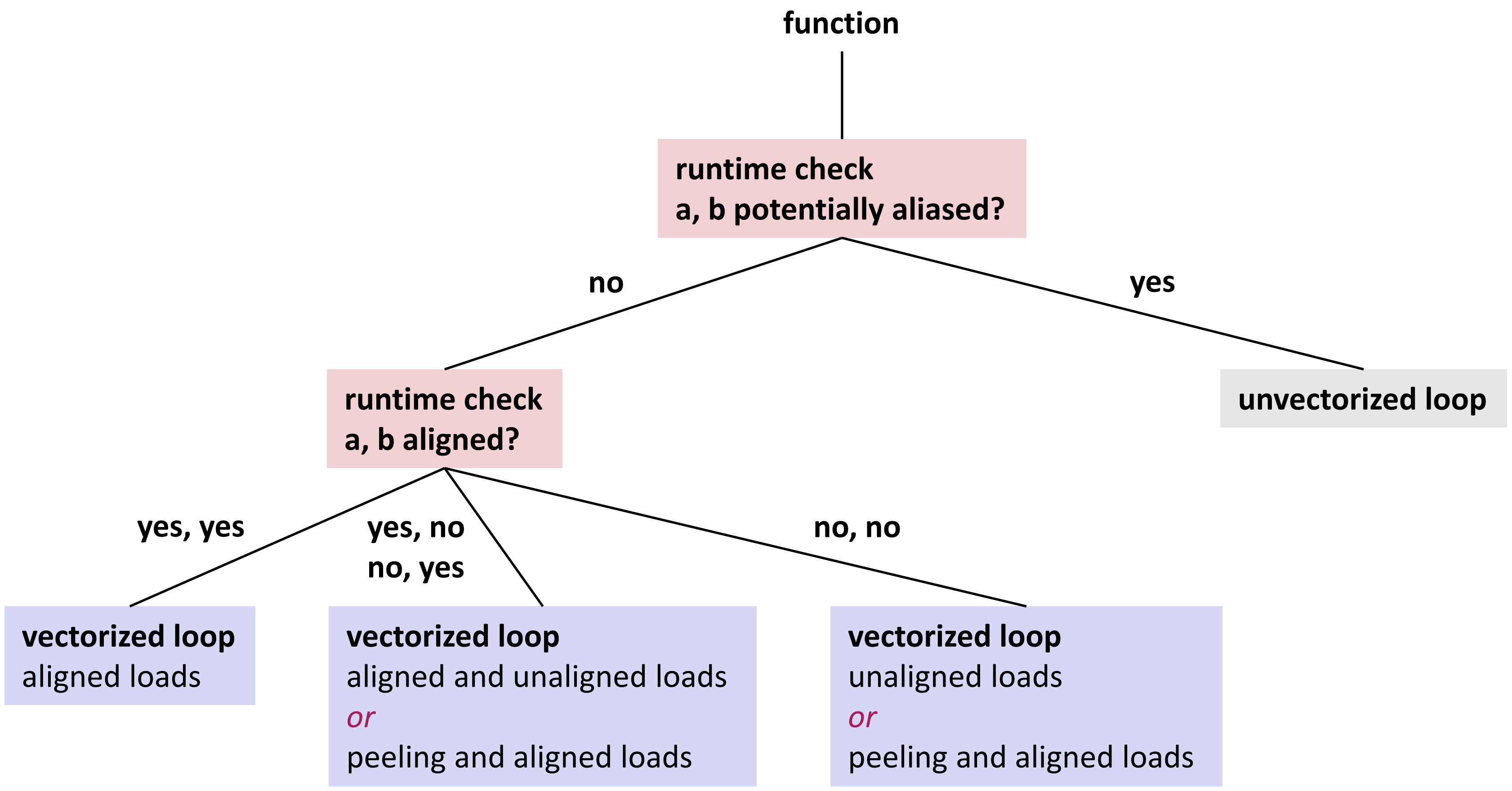
```
void myadd(float *a, float *b, const int n) {
  for (int i = 0; i< n; i++)
    a[i] = a[i] + b[i];
}
```

Assume:

- **No aliasing information**
- **No alignment information**

Can compiler vectorize?

Yes: Through versioning



Compiler Vectorization

- Read manual