Multiprocessors

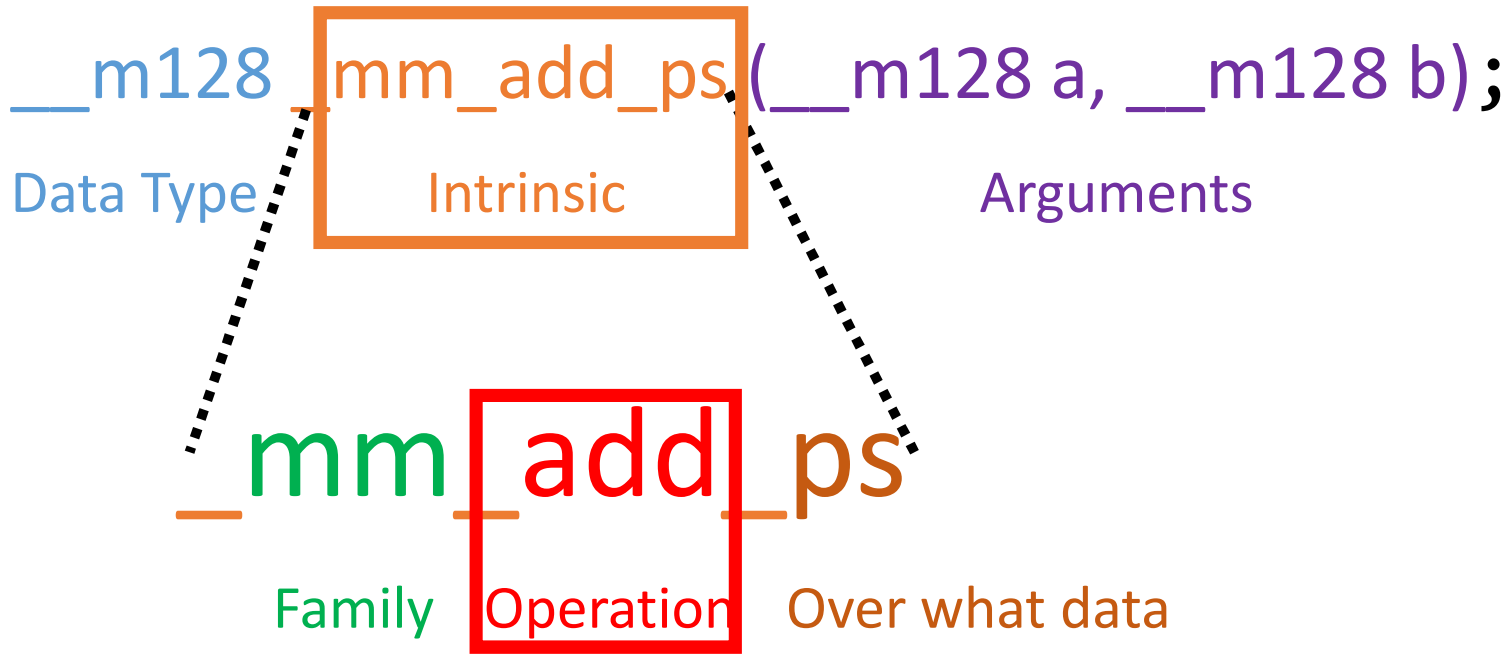
Feb-Jun 2021

4

Vectorization

Alejandro Guajardo Moreno

Lets analyze the instructions…



Instruction Categories

- Natives (translate to a single operation in assembly code)
  - `_mm_load_ps()`
  - `_mm_add_ps()`
  - `_mm_mul_ps()`
  - …

Intrinsic Name	Operation	Corresponding SSE Instruction
`_mm_load_ps`	Load four values, address aligned	`MOVAPS`
`_mm_loadu_ps`	Load four values, address unaligned	`MOVUPS`

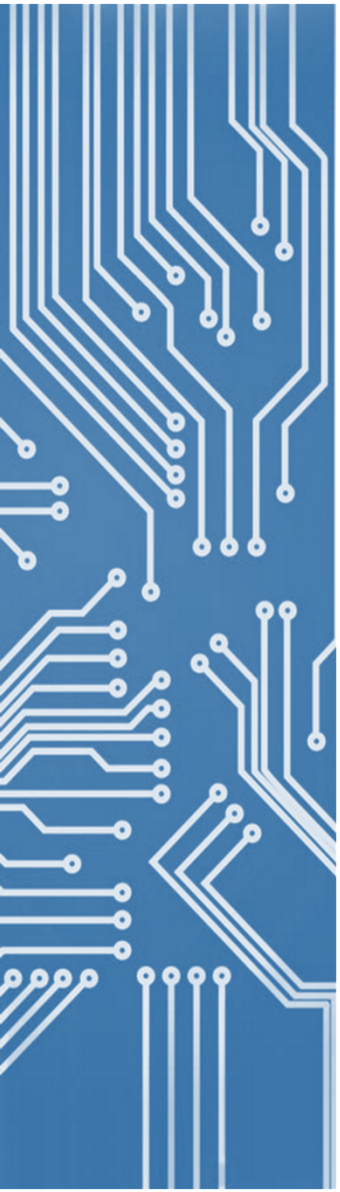
- Multiple (translated to several operations in assembly code)

`_mm_set_ps()`
`_mm_set1_ps()`
…

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

- Macros

`_MM_TRANSPOSEE4_PS()`
`_MM_SHUFFLE()`
…

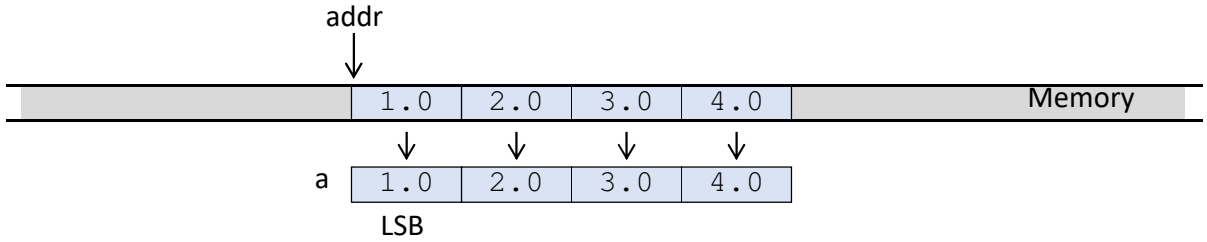


Loads

Intrinsic	Operation	Equivalente instruction (SSE)
_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 DWORDs	MOVSS+shuffling
_mm_load_ps	Load four values, address aligned	MOVAPS
_mm_loadu_ps	Load four values, address unaligned (slow)	MOVUPS
_mm_loadr_ps	Load four values in reverse	MOVAPS+shuffling

Intrinsic	Operation	Equivalente instruction (SSE)
_mm_set_ss	Assign the low DWORD, zeroing the 3 higher ones	Múltiple
_mm_setl_ps	Assign the same value to the 4 DWORDs of a register	Múltiple
_mm_set_ps	Assign 4 DWORD values to a register	Múltiple
_mm_setr_ps	Assign 4 DWORD values to a registar, in reverse order	Múltiple
_mm_setzero_ps	Zero the 4 DWORDs of a register	Múltiple

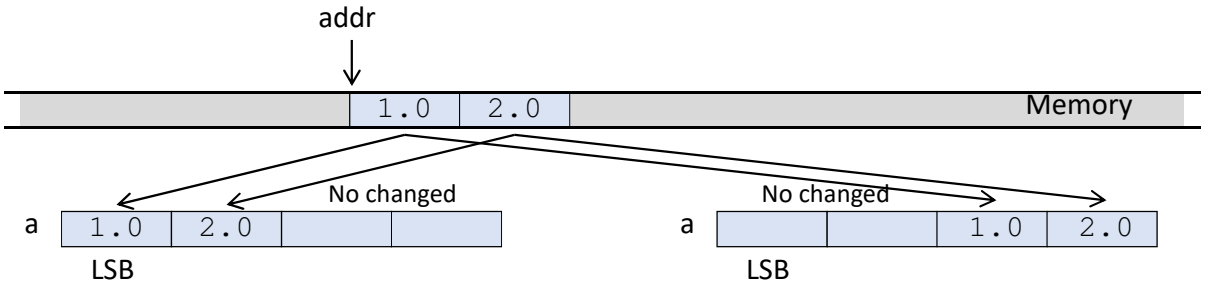
Loads




```
__m128 a = _mm_load_ps(addr);          //Aligned to a 16-byte border
__m128 a = _mm_loadu_ps(addr);         //Non memory aligned (slower)
```

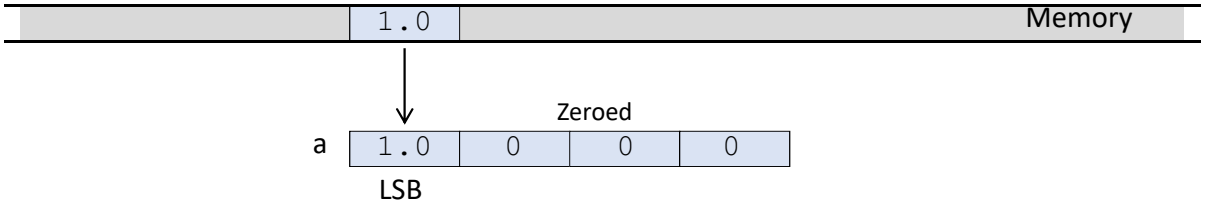
- Memory alignment is important for the efficiency of the intrinsics (because of the hardware).
- `_m128`, `_m128d`, `_m128i` declarations are always aligned to a 16 byte frontier.
- `_m256`, `_m256d`, `_m256i` declarations are always aligned to a 32 byte frontier.
- For array, a directive will be required: **`__declspec(align(16))`** `float g[4];`
- For dynamic memory use `_mm_alloc()` and `_mm_free()`

Loads

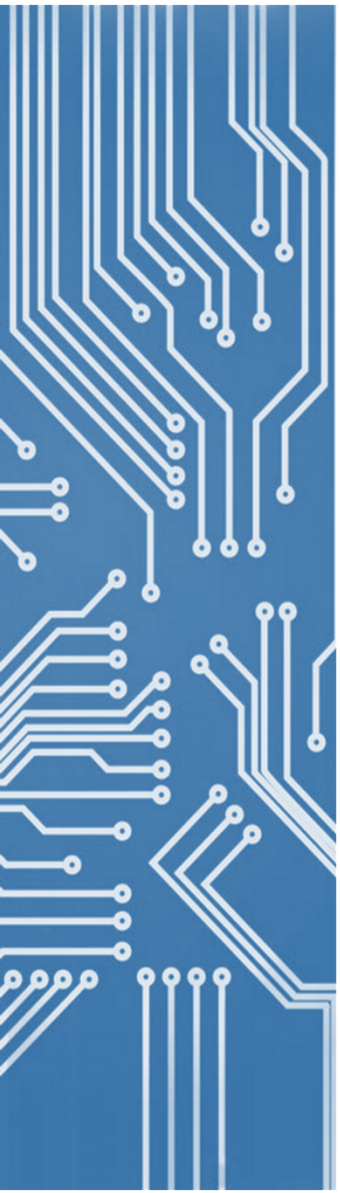



```
a = _mm_loadl_pi(addr);
```

```
a = _mm_loadh_pi(addr);
```




```
a = _mm_load_ss(addr);
```

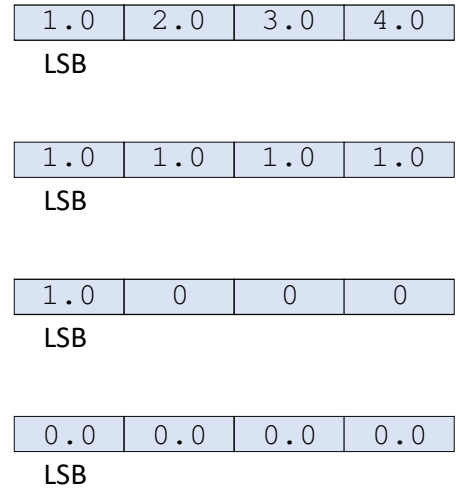


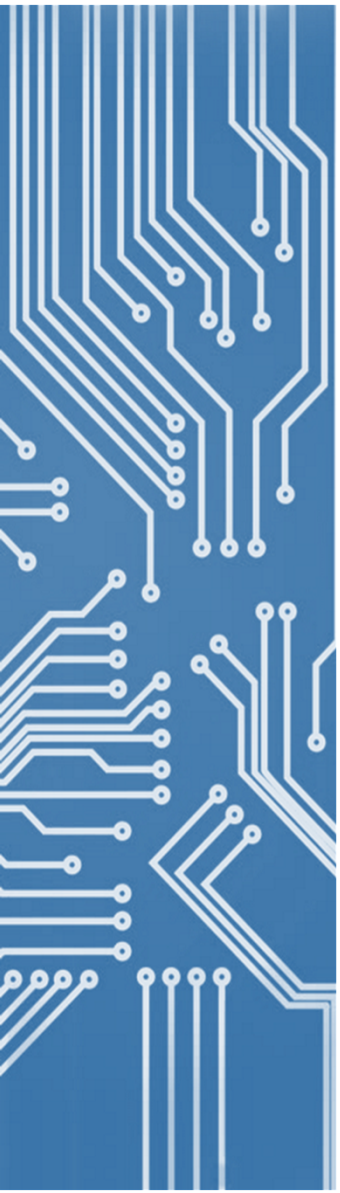
Stores

The operations to store data are similar to the load ones

Intrinsic	Operation	Equivalent instruction (SSE)
_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 DWORDs, 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


1.0 2.0 3.0 4.0
a
a = _mm_set_ps(4.0, 3.0, 2.0, 1.0)
LSB
1.0 1.0 1.0 1.0
b
b = _mm_set1_ps(1.0)
LSB
1.0 0 0 0
c
c = _mm_set_ss(1.0)
LSB
0.0 0.0 0.0 0.0
d
d = _mm_setzero_ps()
LSB



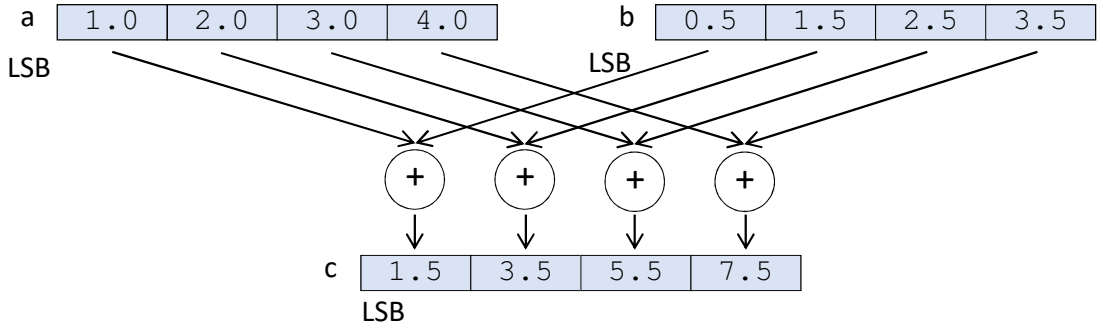
Arithmetics

Intrinsic	Operation	Equivalent instruction (SSE)
_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	Square root	SQRTSS
_mm_sqrt_ps	Square 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

Intrinsic	Operation	Equivalent instruction (SSE)
_mm_addsub_ps	Add and subtract	ADDSUBPS
_mm_hadd_ps	Addition	HADDPS
_mm_hsub_ps	Subtraction	HSUBPS

Intrinsic	Operation	Equivalent instruction (SSE)
_mm_dp_ps	Dot product, simple precision	DPPS

Arithmetics




```
c = _mm_add_ps(a,b);
...
c = _mm_sub_ps(a,b);
c = _mm_mul_ps(a,b);
...
```

Addition Example

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

Variable i is a scalar … conversion to vector will be required.

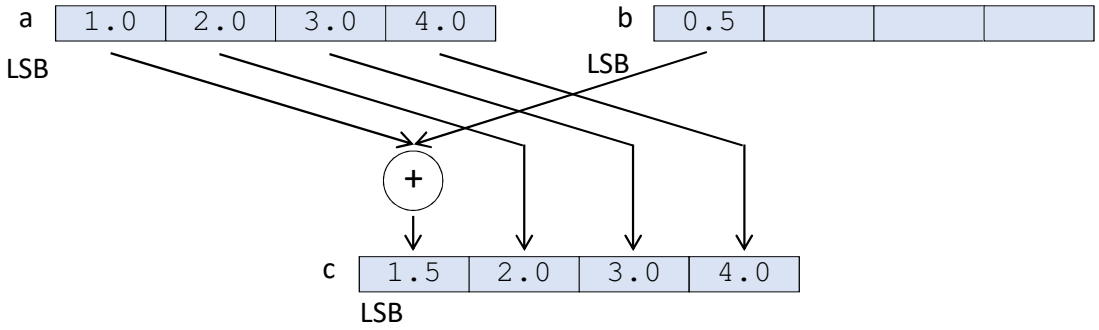
```
#include <intrin.h>
//Consider n multiple of 4, x memory aligned
void addindex_vec(float *x, int n) {
    __m128 index, x_vec;
    for(int i=0; i<n/4; i++) {
      //Load vector x
        x_vec = _mm_load_ps(x+i*4);
       //Initialize vector with values from i
       index = _mm_set_ps(i*4+3, i*4+2, i*4+1, i*4);
       //Add: x[i]+i … but using vectors
       x_vec = _mm_add_ps(x_vec, index);
       // add the two
       _mm_store_ps(x+i*4, x_vec);
    }
}
```

Addition Example (another option)

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

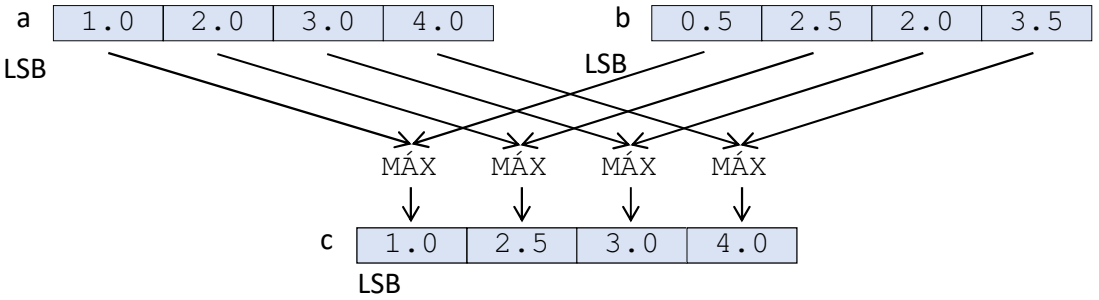
```
#include <intrin.h>

void addindex_vec(float* x, int n) {
    __m128 index, incr, x_vec;
    index = _mm_set_ps(3.0, 2.0, 1.0, 0.0);
    incr = _mm_set1_ps(4.0);
    for (int i=0; i<n/4; i++) {
        x_vec = _mm_load_ps(x + i*4);
        x_vec = _mm_add_ps(x_vec, index);
        _mm_store_ps(x + i*4, x_vec);
        index = _mm_add_ps(index, incr);
    }
}
```

Arithmetics



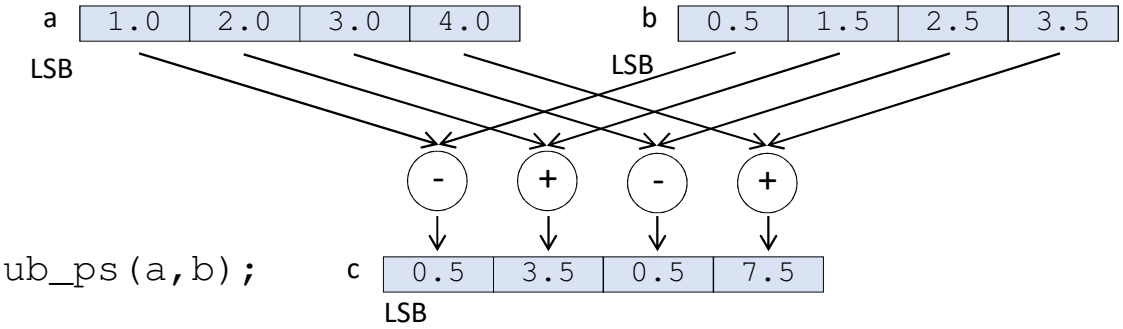

```
c = _mm_add_ss(a,b);
```

Arithmetics

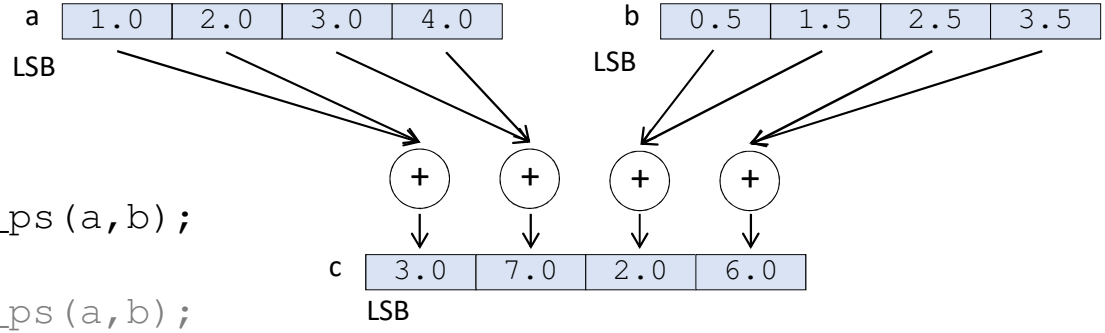



```
c = _mm_max_ps(a,b);
```

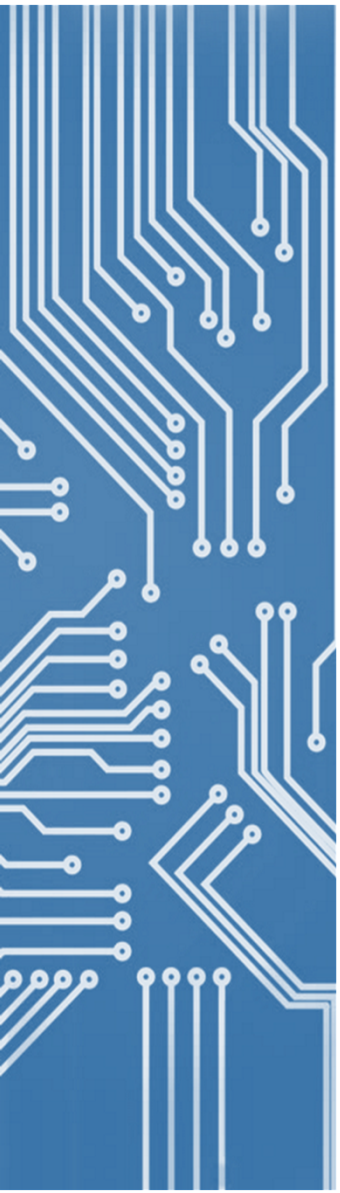
Arithmetics




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




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



Teams

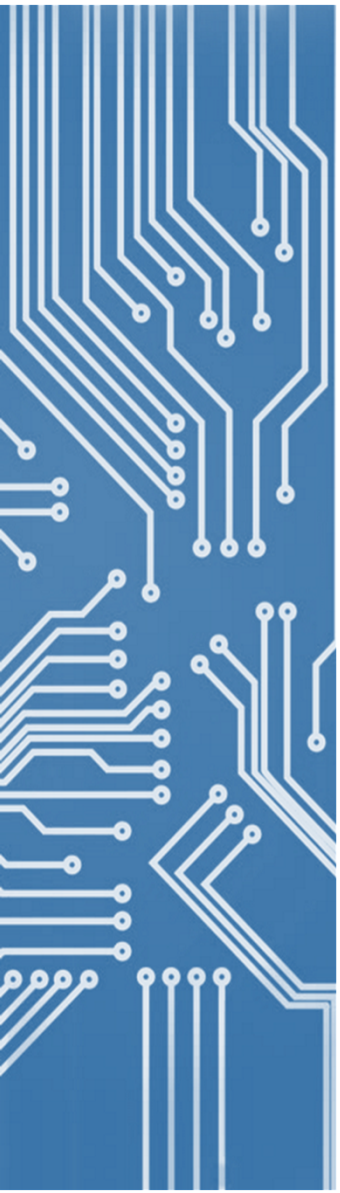
4

What does this code do?

```
//Consider n multiple of 8, x y y memory aligned
void lp(float* x, float* y, int n) {
    __m128 div, v1, v2, avg;
    div = _mm_set1_ps(2.0);
    for (int i = 0; i < n / 8; i++) {
        v1 = _mm_load_ps(x + i * 8);       //??
        v2 = _mm_load_ps(x + 4 + i * 8);   //??
        avg = _mm_hadd_ps(v1, v2);         //??
        avg = _mm_div_ps(avg, div);        //??
        _mm_store_ps(y + i * 4, avg);      //??
     }
}
```

Partial reduction example (Average)

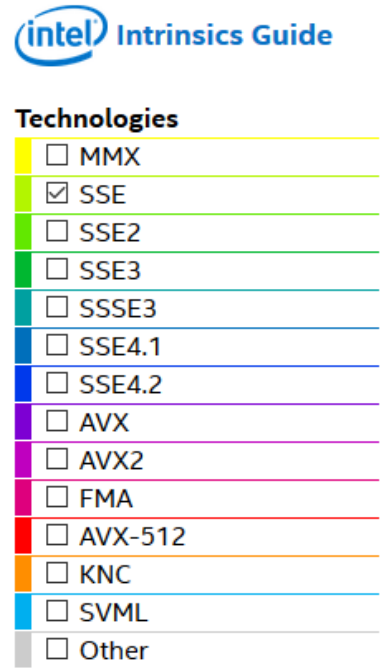
```
//Consider n multiple of 8, x y y memory aligned
void lp(float* x, float* y, int n) {
    __m128 div, v1, v2, avg;
    div = _mm_set1_ps(2.0);
    for (int i = 0; i < n / 8; i++) {
        v1 = _mm_load_ps(x + i * 8);       //Load the first 4 numbers
        v2 = _mm_load_ps(x + 4 + i * 8);   //Load the second 4 numbers
        avg = _mm_hadd_ps(v1, v2);         //Add pairs
        avg = _mm_div_ps(avg, div);        //Average the two numbers
added in the previous step by dividing by 2.0
        _mm_store_ps(y + i * 4, avg);      //Store in memory
    }
}
```



Teams

1

Using only SSE Intrinsics, increase the performance of this code.

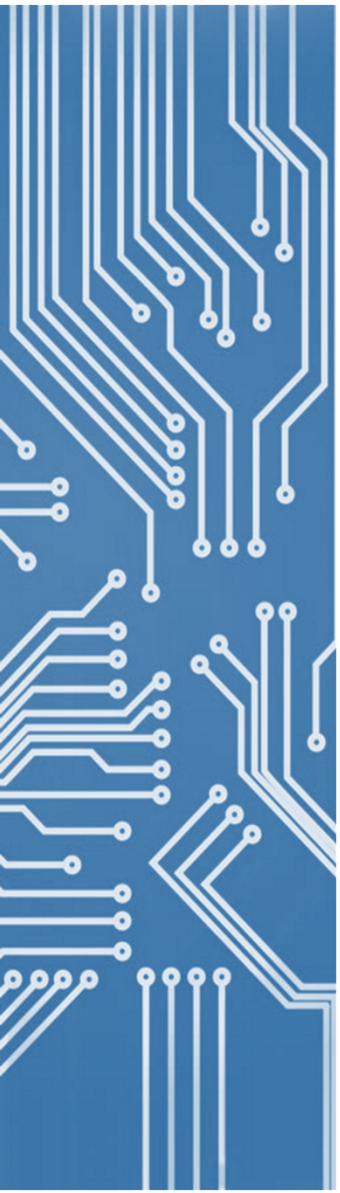



```
//Consider n multiple of 8, x and y memory aligned
void lp(float* x, float* y, int n) {
    __m128 div, v1, v2, avg;
    div = _mm_set1_ps(2.0);
    for (int i = 0; i < n / 8; i++) {
        v1 = _mm_load_ps(x + i * 8);       //Load the first 4 numbers
        v2 = _mm_load_ps(x + 4 + i * 8);   //Load the second 4 numbers
        avg = _mm_hadd_ps(v1, v2);         //Add pairs
        avg = _mm_div_ps(avg, div);        //Average the two numbers
added in the previous step by dividing by 2.0
        _mm_store_ps(y + i * 4, avg);      //Store in memory
    }
}
```

Solution…

Given that:

$$\frac{a+b}{2} = (a+b) * \left(\frac{1}{2}\right) = (a+b) * (0.5)$$



Solution…

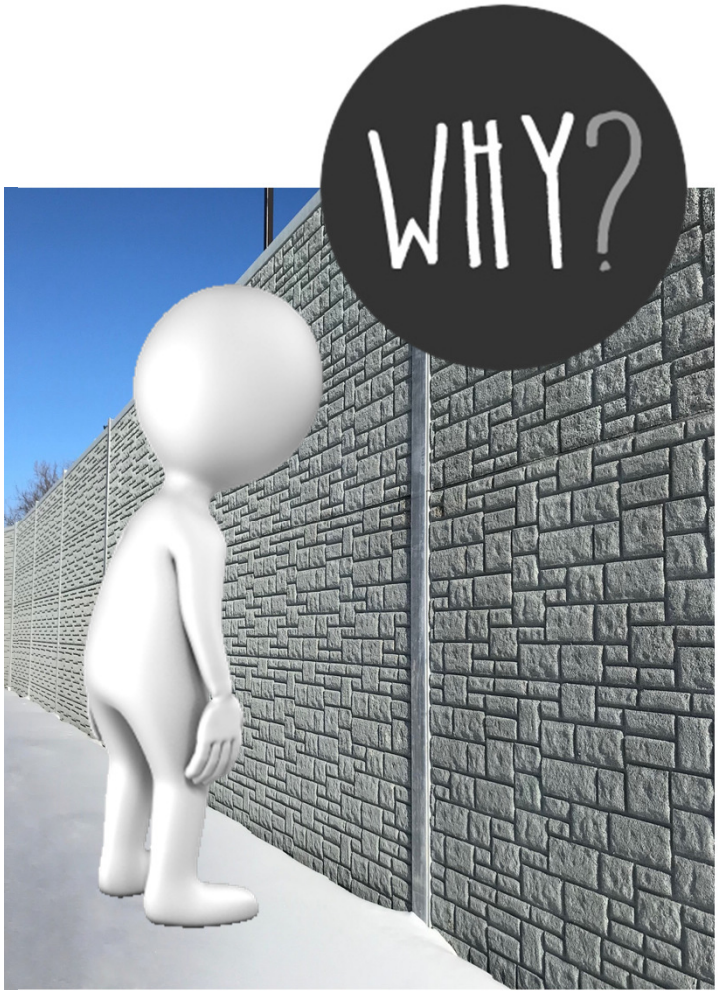
Division

```
void lp(float* x, float* y, int n) {
   __m128 div, v1, v2, avg;
   div = _mm_set1_ps(2.0);
   for (int i = 0; i < n / 8; i++) {
      v1 = _mm_load_ps(x + i * 8);
      v2 = _mm_load_ps(x + 4 + i * 8);
      avg = _mm_hadd_ps(v1, v2);
      avg = _mm_div_ps(avg, div);
      _mm_store_ps(y + i * 4, avg);
    }
}
```

Multiplication

```
void lp(float* x, float* y, int n) {
   __m128 half, v1, v2, avg;
   half = _mm_set1_ps(0.5);
   for (int i = 0; i < n / 8; i++) {
      v1 = _mm_load_ps(x + i * 8);
      v2 = _mm_load_ps(x + 4 + i * 8);
      avg = _mm_hadd_ps(v1, v2);
      avg = _mm_mul_ps(avg, half);
      _mm_store_ps(y + i * 4, avg);
   }
}
```

Solution…



Division vs. Multiplication

`__m128 _mm_div_ps (__m128 a, __m128 b)`

Synopsis

```
__m128 _mm_div_ps (__m128 a, __m128 b)
#include <xmmintrin.h>
Instruction: divps xmm, xmm
CPUID Flags: SSE
```

Description

Divide packed single-precision (32-bit) floating-point elemen

Operation

```
FOR j := 0 to 3
        i := 32*j
        dst[i+31:i] := a[i+31:i] / b[i+31:i]
ENDFOR
```

Performance

Architecture	Latency	Throughput (CPI)
Knights Landing	38	~10
Skylake	11	3
Broadwell	<11	4
Haswell	<13	6
Ivy Bridge	14-Nov	6

`__m128 _mm_mul_ps (__m128 a, __m128 b)`

Synopsis

```
__m128 _mm_mul_ps (__m128 a, __m128 b)
#include <xmmintrin.h>
Instruction: mulps xmm, xmm
CPUID Flags: SSE
```

Description

Multiply packed single-precision (32-bit) floating-point eleme

Operation

```
FOR j := 0 to 3
        i := j*32
        dst[i+31:i] := a[i+31:i] * b[i+31:i]
ENDFOR
```

Performance

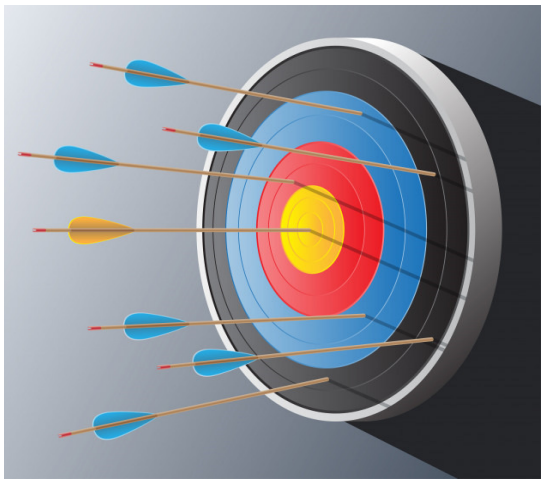
Architecture	Latency	Throughput (CPI)
Skylake	4	0.5
Broadwell	3	0.5
Haswell	5	0.5
Ivy Bridge	5	1

Division vs. Multiplication

Division

Accuracy

Multiplication

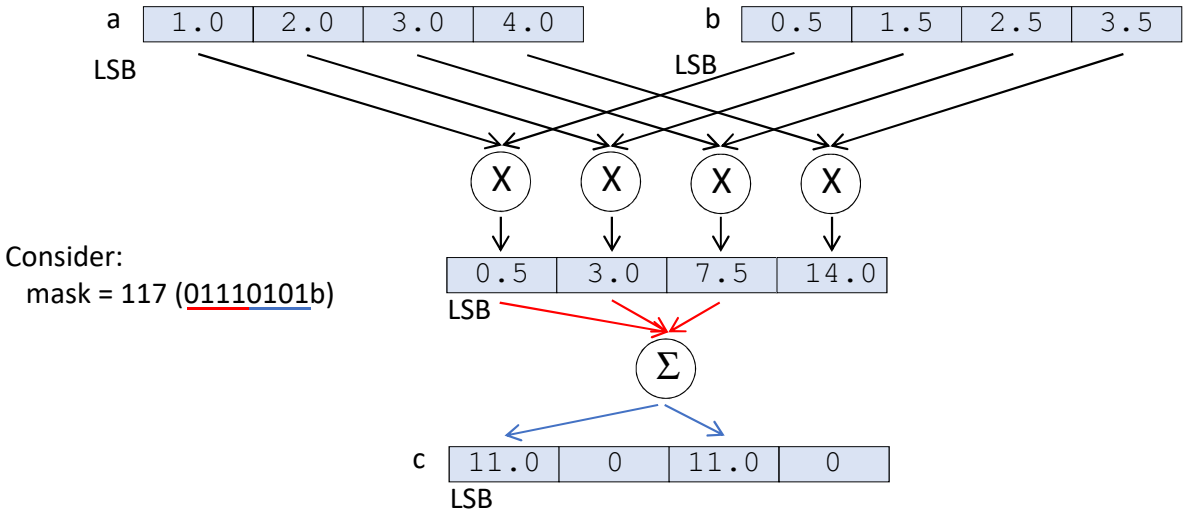


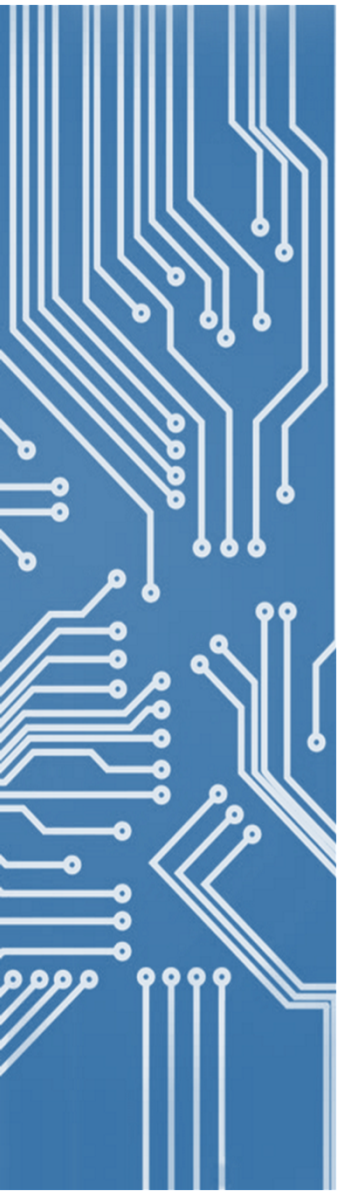
Speed

Arithmetic (dot product)

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

(SSE4) Calculates the dot product of two vectors, considering only the selected terms … un selected terms will set to zero. Term selection depends on a bit mask that goes as the last parameter of the calling.



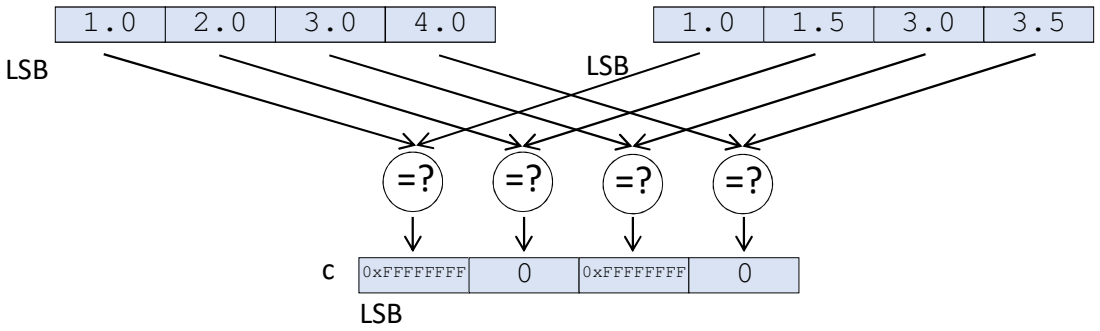


Comparisons

Intrinsic	Operation	Eq. Inst. (SSE3)
_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	CMPGTSS
_mm_cmpgt_ps	Greater than	CMPGTPS
_mm_cmpge_ss	Greater than or equal	CMPGESS
_mm_cmpge_ps	Greater than or equal	CMPGEPS
_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	CMPLESS
_mm_cmpngt_ps	Not greater than	CMPLEPS

Intrinsic	Operation	Eq. Inst. (SSE3)
_mm_cmpnge_ss	Not greater than or equal	CMPLTSS
_mm_cmpnge_ps	Not greater than or equal	CMPLTPS
_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	Greather than	COMISS
_mm_comige_ss	Greater than or equal	COMISS
_mm_comineq_ss	Not equal	COMISS
_mm_ucomineq_ss	Equal	UNCOMISS
_mm_ucomilt_ss	Less than	UNCOMISS
_mm_ucomile_ss	Less than or equal	UNCOMISS
_mm_ucomigt_ss	Greater than	UNCOMISS
_mm_ucomige_ss	Greater than or equal	UNCOMISS
_mm_ucomineq_ps	Not equal	UNCOMISS

Comparisons

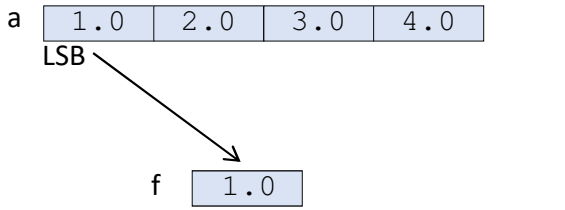


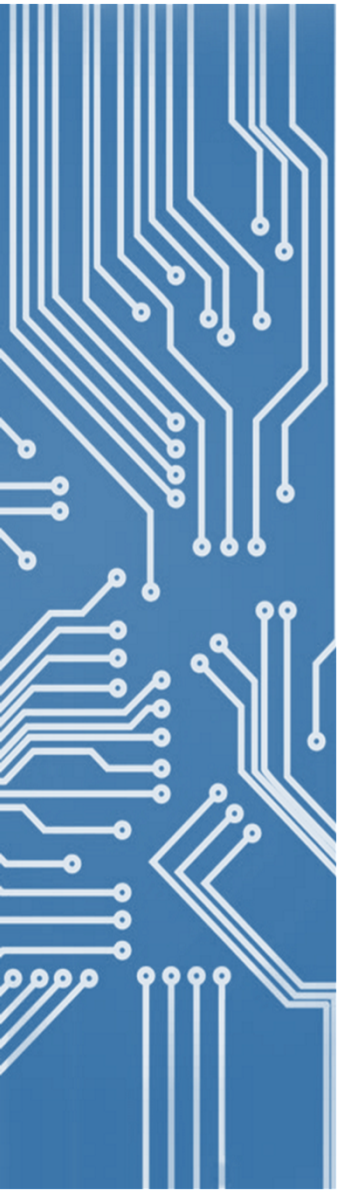

```
c = _mm_cmpeq_ps(a,b);
...
c = _mm_cmple_ps(a,b);
c = _mm_cmplt_ps(a,b);
c = _mm_cmpge_ps(a,b);
...
```

Conversions

Intrinsic	Operation	Equivalent Instruction (SSE3)
_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 to32-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

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





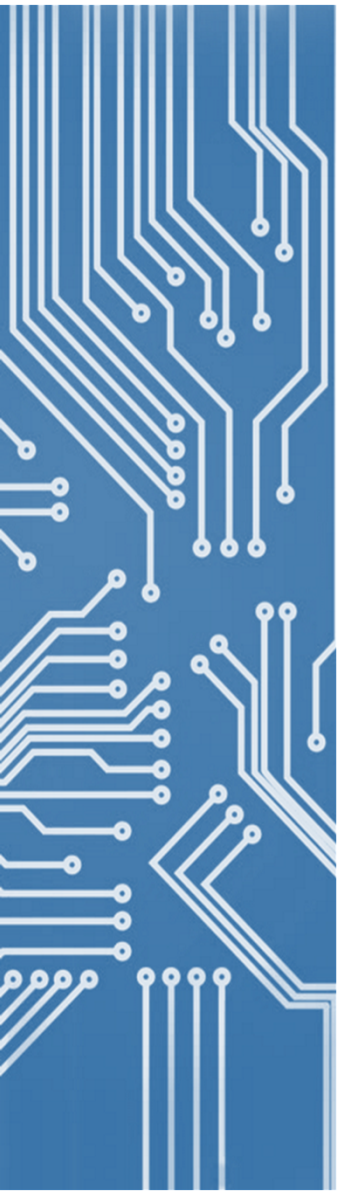
Shuffles

Intrinsic	Operation	Equivalent Instruction (SSE3)
_mm_shuffle_ps	Shuffle	SHUFPS
_mm_unpackhi_ps	Unpacks high	UNPCKHPS
_mm_unpacklo_ps	Unpacks low	UNPCKLPS
_mm_move_ss	Assign low part and discard the remaining	MOVSS
_mm_movehl_ps	Moves high to low	MOVHLPS
_mm_movelh_ps	Moves low to high	MOVLHPS
_mm_movemask_ps	Creates four-bit mask	MOVMSKPS

Intrinsic	Operation	Equivalent Instruction (SSE3)
_mm_movehdup_ps	Duplicates	MOVSHDUP
_mm_moveldup_ps	Duplicates	MOVLHDUP

Intrinsic	Operation	Equivalent Instruction (SSE3)
_mm_shuffle_epi8	Shuffle	PSHUFP
_mm_alignr_epi8	Shift	PALIGNR

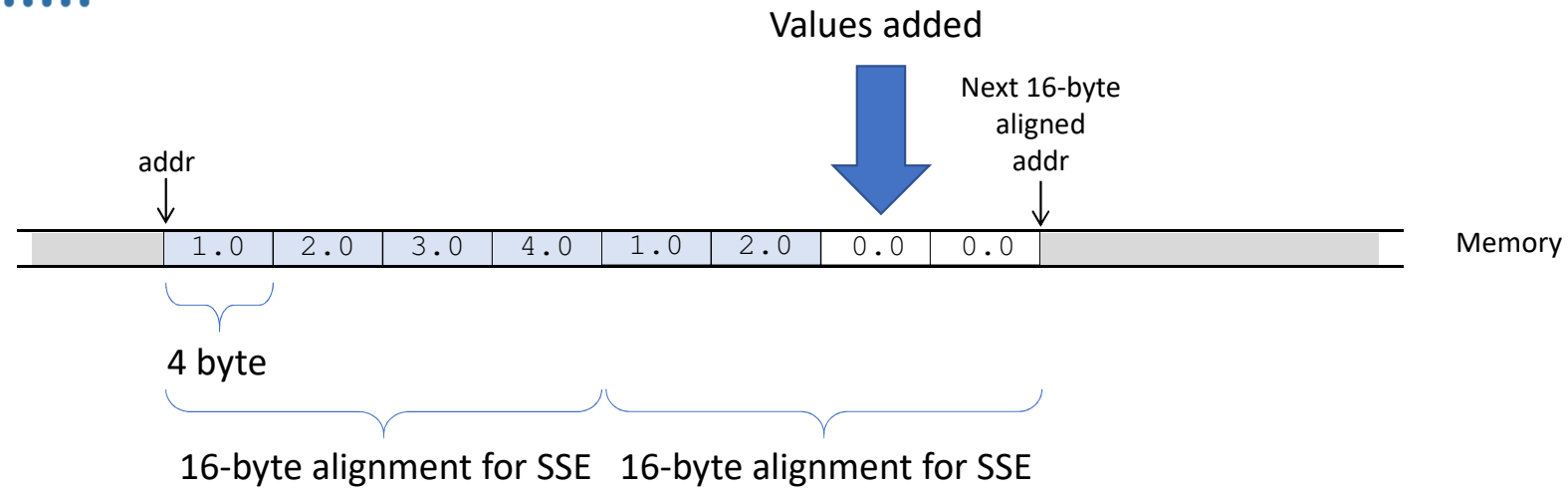
Intrinsic	Operation	Equivalent Instruction (SSE3)
__m128 _mm_blend_ps	Selects single precision float data from two sources using constant mask.	BLENDPS
__m128 _mm_blendv_ps	Selects single precision float data from two sources using variable mask.	BLENDVPS
__m128 _mm_insert_ps	Insert single precision float into packed single precision array element selected by index.	INSERTPS
int _mm_extract_ps	Extract single precision float from packed single precision array element selected by index.	EXTRACTPS



Other functions

- Logics (AND, OR, XOR, NOT, …)
- Cache support:
  - `_mm_prefecth(···)`
  - `_mm_stream_ps(···)`
- … and many others.

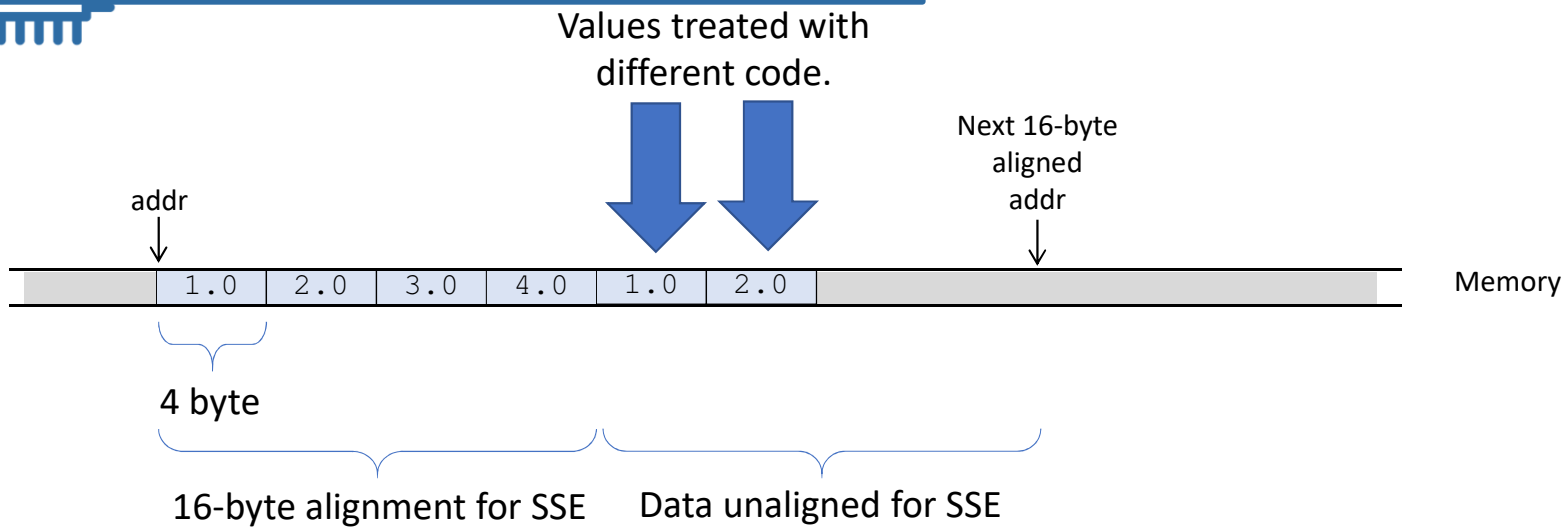
Padding



Padding:

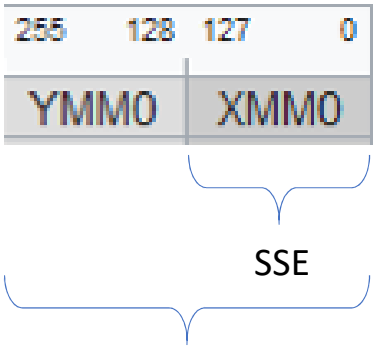
- Allows the correct alignment of data in memory for use with VECTORS.
- The values added will not affect the outcome of the calculation.

Peeling



Peeling:
- Allows the aligned data in memory to be used with VECTORS.
- The additional code increases the complexity of the program.

Performance Notes on SSE ⇔ AVX



- XMM and YMM share the lower 128 bits.
- Transitioning between AVX and SSE can lead to undefined values in the upper 128bits.
- Compiler saves the upper 128bits, clear it, execute the old SSE operation, and then restore the old value.
- Overhead to AVX operations, resulting in reduced performance.
- The transition penalty is when non-VEX __m128 instructions are combined with __m256 instructions.
- This happens when you use old SSE libraries linked into new AVX enabled programs.

✓ If really needed, use AVX ‘s new instruction set for __m128, with VEX prefixes to avoid performance penalties.

✓ Don´t mix AVX and SSE.

Instruction Set Extensions Cheat Sheet…

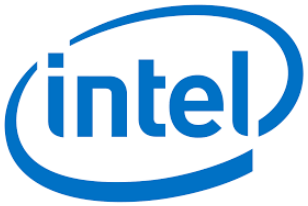
Instruction Sets

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

I.Set	Intel *(Year)*	AMD *(Year)*	Description
x86	all	all	x86 Base Instructions
SSE	Pentium III *99*	K7 Palomino *01*	Streaming SIMD Extensions
SSE2	Pentium 4 *01*	K8 *03*	
SSE3	Prescott *04*	K9 *05*	
SSSE3	Woodcrest *06*	Bulldozer *11*	
SSE4.1	Penryn *07*	Bulldozer *11*	
SSE4.2	Nehalem *08*	Bulldozer *11*	
AVX	Sandy Bridge *11*	Bulldozer *11*	Advanced Vector Extensions
AVX2	Haswell *13*	-	
FMA	Haswell *13*	Bulldozer *11*	Fused Multiply 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*	-	

Source: finis@in.tum.de

Suggested Material

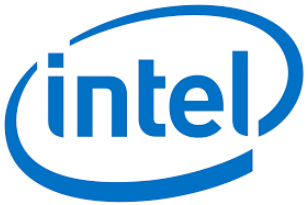


Intel® C++ Intrinsic Reference

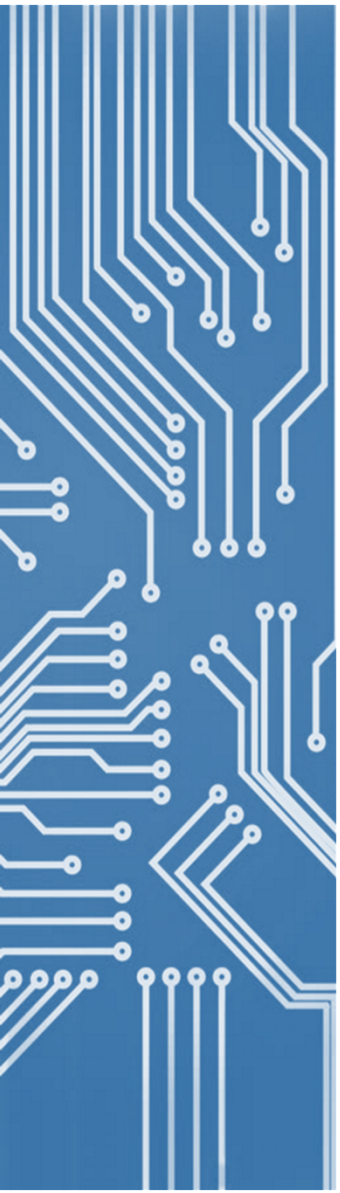
Document Number: 312482-003US

x86 Intrinsics Cheat Sheet

Jan Finis
finis@in.tum.de



https://techdecoded.intel.io/topics/code-modernization/



Lets analyze the instructions…

```
__m128 _mm_add_ps(__m128 a, __m128 b);
```

Data Type — Intrinsic — Arguments

_mm_add_ps

Family — Operation — Over what data

p | **s**

presentation | type

Kahoot!

Nickname: Axxxxxx3-4Iniciales

For Thursday…

Understand why Memory Alignment is important. Read the following:
https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization
https://en.wikipedia.org/wiki/Data_structure_alignment
https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/aligned-malloc?view=vs-2019
Look for the file “Single Threaded Array Addition (FLOATS)s” in Canvas.
1.- Document the time it took to process the loop BEFORE changing the code. Run 5 times the program. Average the 5 results. This will be your BASELINE for comparisons (The ## in “Results verified!!! (##)”).
2.- Rewrite the loop commented “//This loop can be optimized using Intrinsics” using Intrinsics. First use SSE (128-bit) Instruction Set Extensions. Run it 5 times and average the runs. Document the time it took to process the loop.
3.- Change again the loop and now use AVX (256-bit) Instruction Set Extensions. Run 5 times, average and document.
4.- Upload to the platform a Word or PDF Document which includes:
- 1 Table comparing the times it took to run each case and the % of improvement vs. your BASELINE.
- Your conclusions.
5.- Upload your Visual Studio Solution or GCC's code of both SSE and AVX (You can add a menu to the program to choose which code to execute or leave commented one of the codes).
Note: Use what you learned on Memory Alignment. You may have to use __declspec(align()) when defining your variables, depending on the Intrinsics used.
Note 2: You might want to use the Intrinsics: _load_ps(), _loadu_ps(), _store_ps() or _storeu_ps().

Thank you!

SEE YOU NEXT TIME