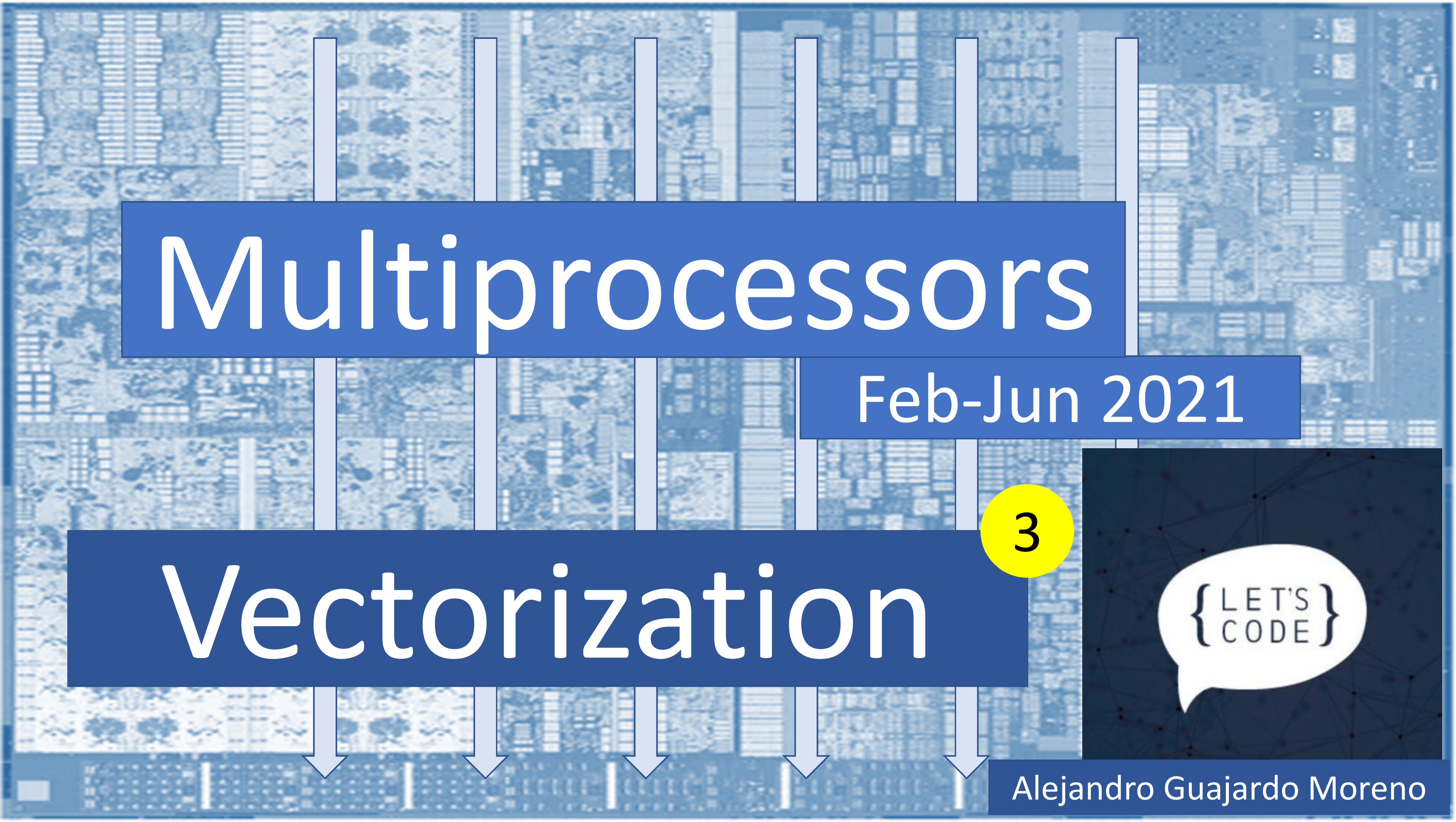


Multiprocessors

Feb-Jun 2021

Vectorization

3

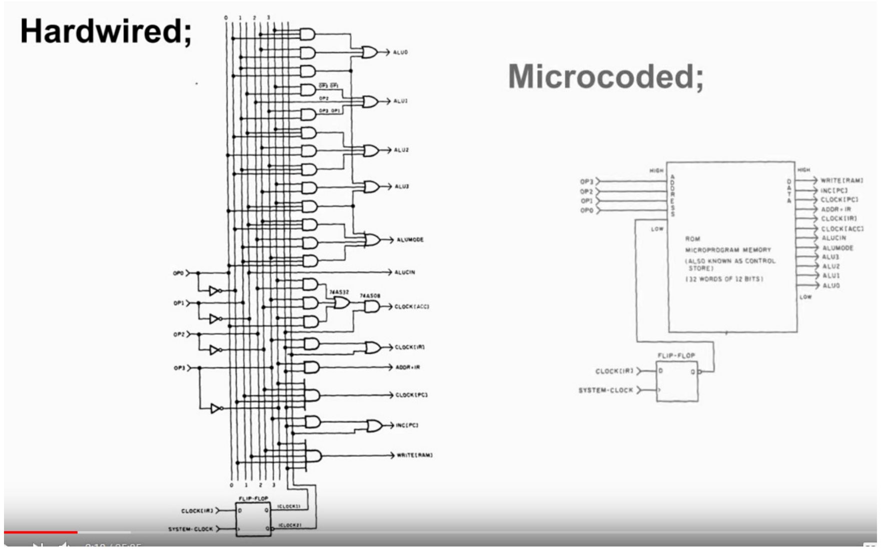
Alejandro Guajardo Moreno

Cyrix…

What Happened to Cyrix Processors? | Nostalgia Nerd

https://www.youtube.com/watch?v=iWGAdoMz1c0

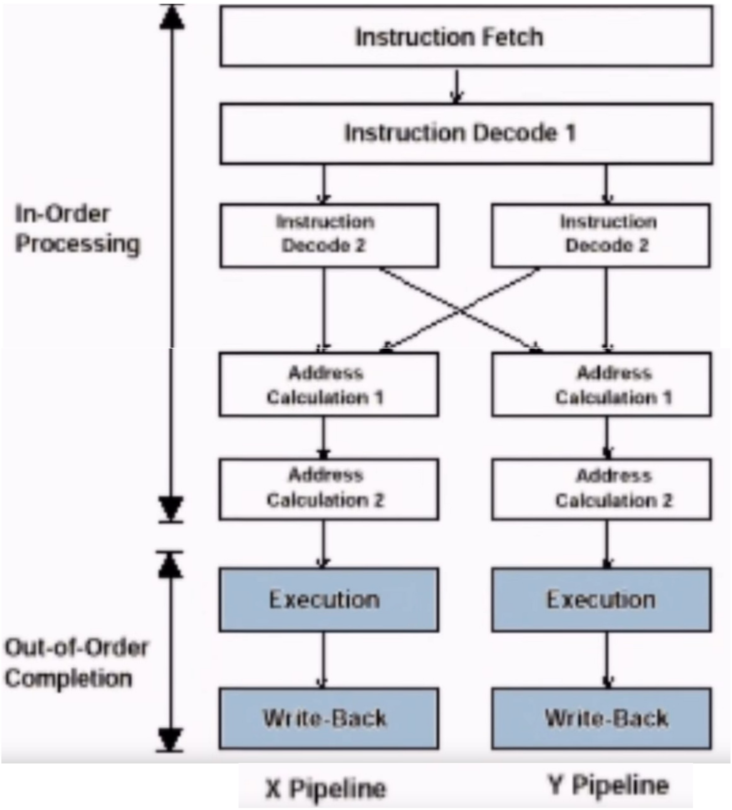
Summary...



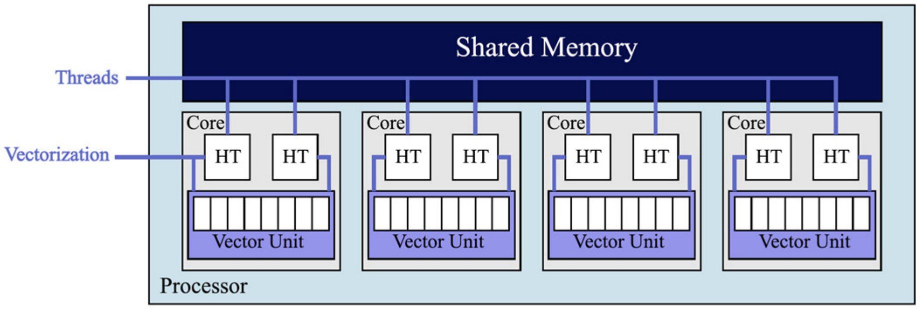
AMD

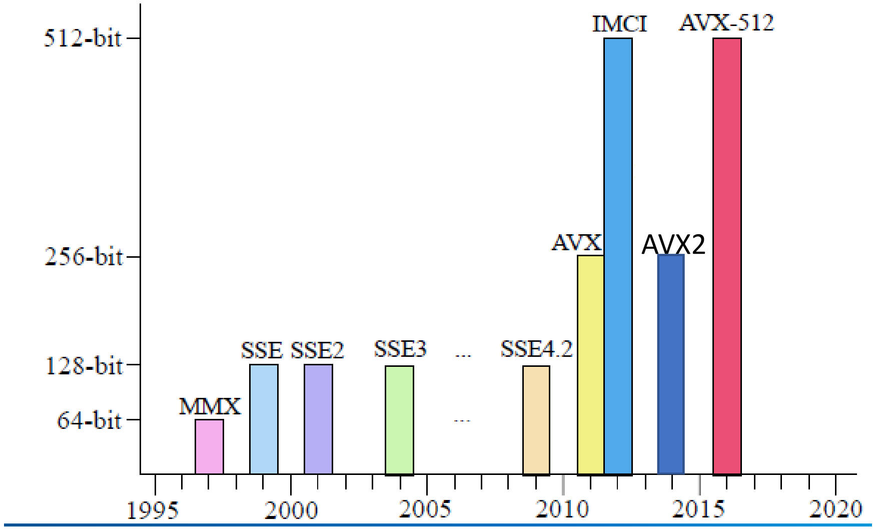
intel

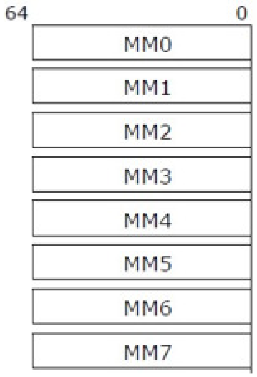
Cyrix



Review from last class...







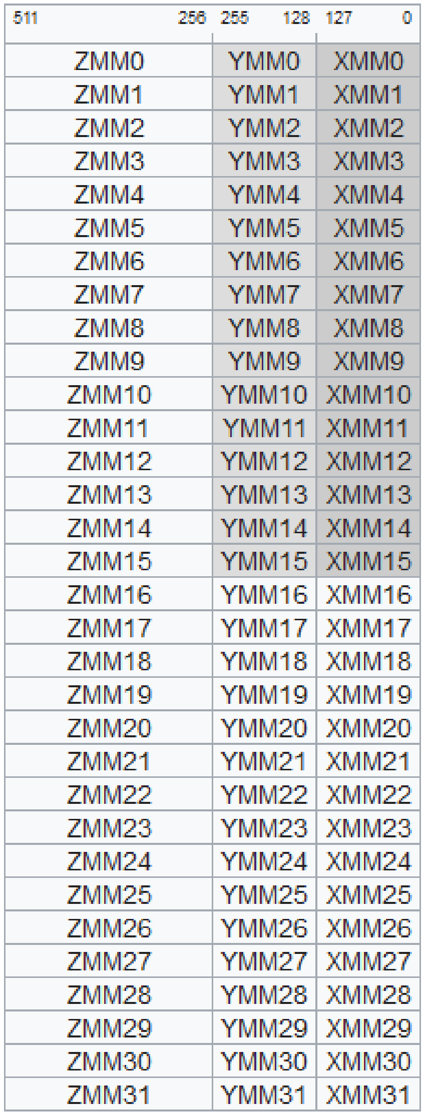
64	0
MM0	
MM1	
MM2	
MM3	
MM4	
MM5	
MM6	
MM7	

MMX

SSE

AVX

AVX-512

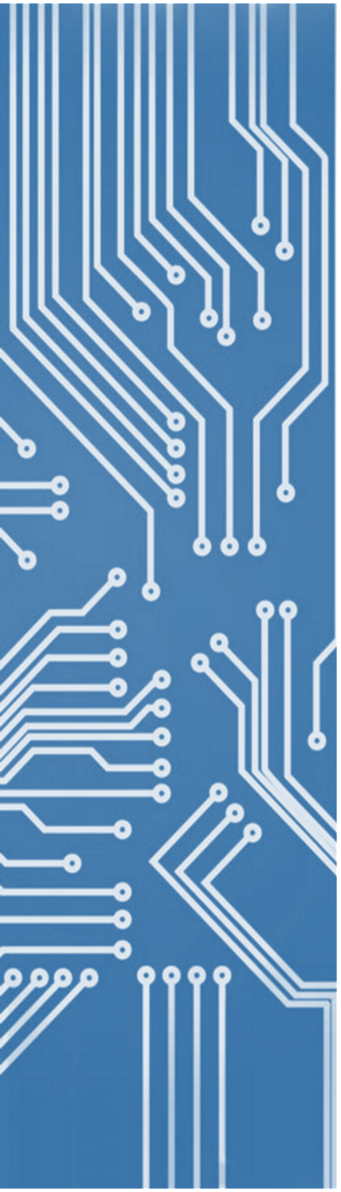


511 256	255 128	127 0
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8
ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15
ZMM16	YMM16	XMM16
ZMM17	YMM17	XMM17
ZMM18	YMM18	XMM18
ZMM19	YMM19	XMM19
ZMM20	YMM20	XMM20
ZMM21	YMM21	XMM21
ZMM22	YMM22	XMM22
ZMM23	YMM23	XMM23
ZMM24	YMM24	XMM24
ZMM25	YMM25	XMM25
ZMM26	YMM26	XMM26
ZMM27	YMM27	XMM27
ZMM28	YMM28	XMM28
ZMM29	YMM29	XMM29
ZMM30	YMM30	XMM30
ZMM31	YMM31	XMM31

Options for vectorization

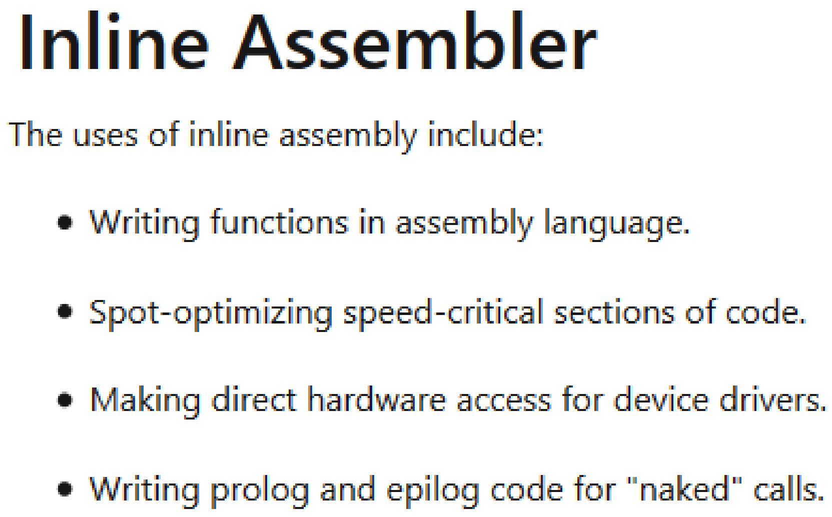
Assembly language

```
..B8.5
  MOVAPS a(,%rdx,4), %xmm0
  ADDPA b(,%rdx,4), %xmm0
  MOVAPS %xmm0, c(,%rdx,4)
  ADDQ $4, %rdx
  CMPQ $rdi, %rdx
  JL ..B8.5
```



Assembly language

Microsoft



Inline Assembler

The uses of inline assembly include:

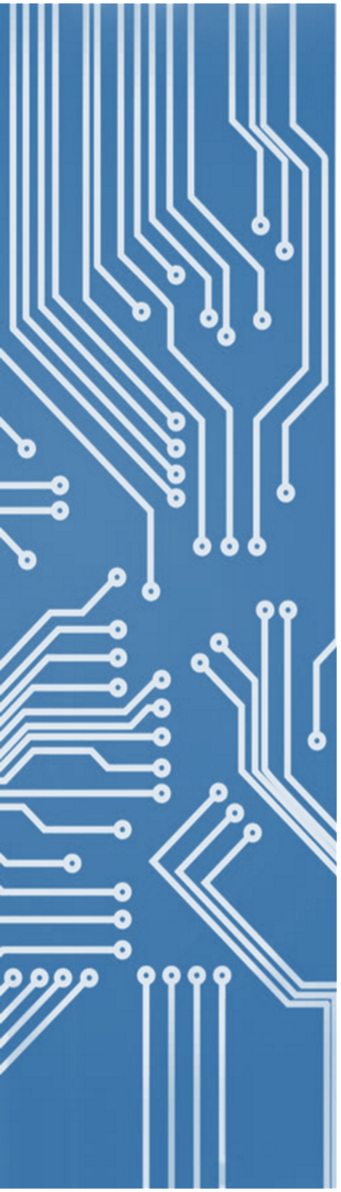
- Writing functions in assembly language.
- Spot-optimizing speed-critical sections of code.
- Making direct hardware access for device drivers.
- Writing prolog and epilog code for "naked" calls.

INTEL® C++ COMPILER

Supports Microsoft-style inline assembly on Windows

https://docs.microsoft.com/en-us/cpp/assembler/inline/inline-assembler?view=vs-2019

Note: Microsoft Macro Assembler (MASM) can also be used.



Assembly language

`__asm` block enclosed in braces:

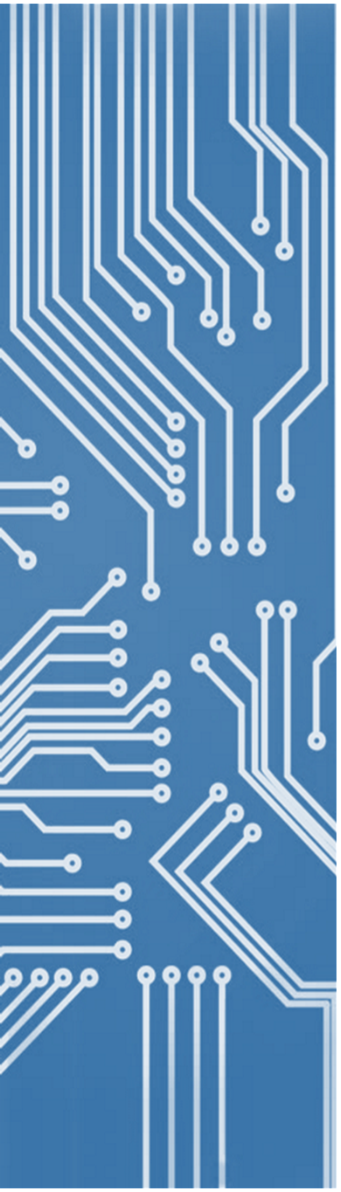
```
__asm {
  mov al, 2
  mov dx, 0xD007
  out dx, al
}
```

`__asm` in front of each assembly instruction:

```
__asm mov al, 2
__asm mov dx, 0xD007
__asm out dx, al
```

you can also put assembly instructions on the same line:

```
__asm mov al, 2   __asm mov dx, 0xD007   __asm out dx, al
```



Options for vectorization

Assembly language

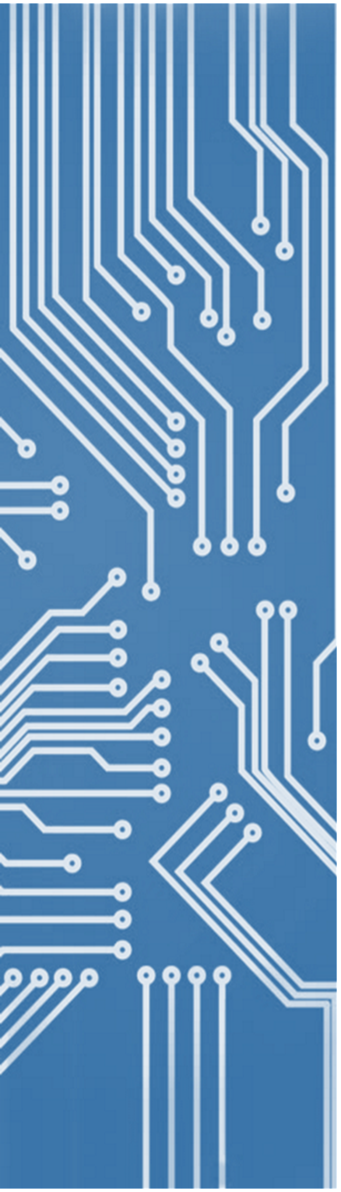
```
..B8.5
  MOVAPS a(,%rdx,4), %xmm0
  ADDPA b(,%rdx,4), %xmm0
  MOVAPS %xmm0, c(,%rdx,4)
  ADDQ $4, %rdx
  CMPQ $rdi, %rdx
  JL ..B8.5
```

Explicit Vectorization

Intrinsic Functions

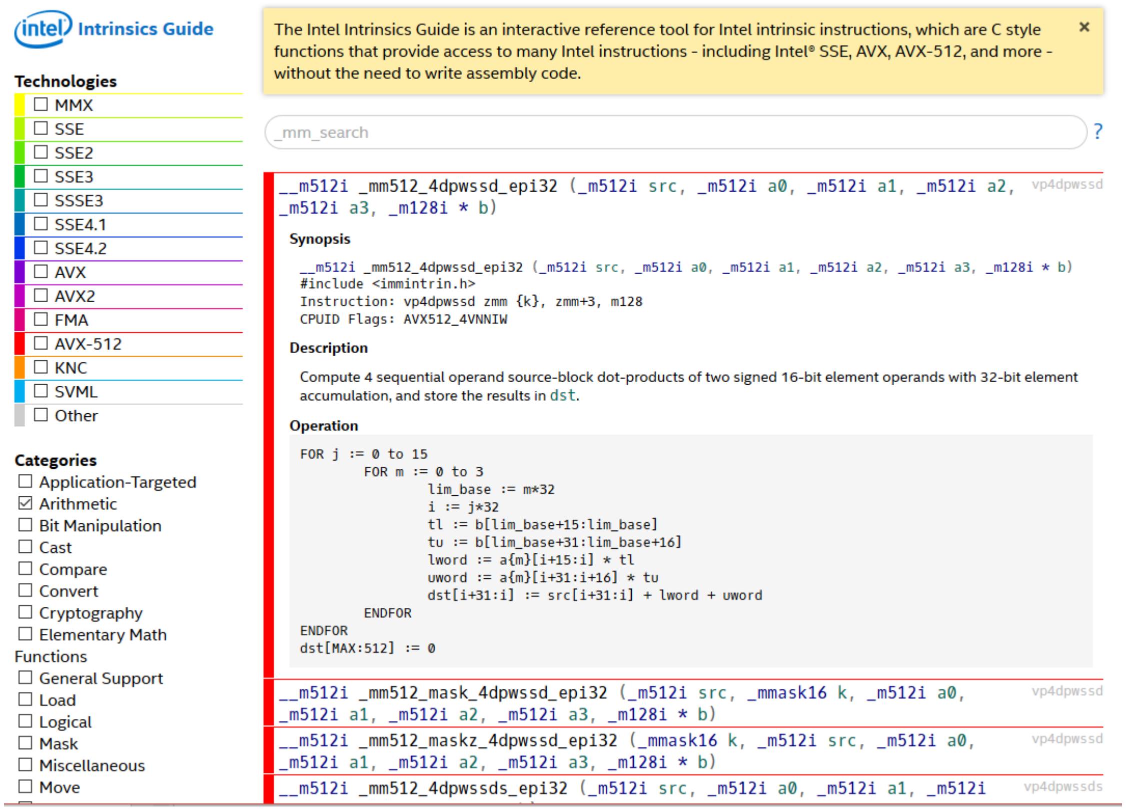
- Middle ground Assembly ⇔ C
- Direct Access to processor instructions
- Retain C Syntax

```
void ejemplo(){
    __m128 rA, rB, rC;
    for (int i = 0; i<LEN; i+=4){
        rA = _mm_load_ps(&a[i]);
        rB = _mm_load_ps(&b[i]);
        rC = _mm_add_ps(rA,rB);
        _mm_store_ps(&c[i], rC);
    }
}
```



Intrinsics

intel Intrinsics Guide



intel Intrinsics Guide

The Intel Intrinsics Guide is an interactive reference tool for Intel intrinsic instructions, which are C style functions that provide access to many Intel instructions - including Intel® SSE, AVX, AVX-512, and more - without the need to write assembly code. ×

Technologies

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☐ AVX
- ☐ AVX2
- ☐ FMA
- ☐ AVX-512
- ☐ KNC
- ☐ SVML
- ☐ Other

Categories

- ☐ Application-Targeted
- ☑ Arithmetic
- ☐ Bit Manipulation
- ☐ Cast
- ☐ Compare
- ☐ Convert
- ☐ Cryptography
- ☐ Elementary Math Functions
- ☐ General Support
- ☐ Load
- ☐ Logical
- ☐ Mask
- ☐ Miscellaneous
- ☐ Move

_mm_search ?

`__m512i _mm512_4dpwssd_epi32 (_m512i src, _m512i a0, _m512i a1, _m512i a2, _m512i a3, _m128i * b)` vp4dpwssd

Synopsis

```
__m512i _mm512_4dpwssd_epi32 (_m512i src, _m512i a0, _m512i a1, _m512i a2, _m512i a3, _m128i * b)
#include <immintrin.h>
Instruction: vp4dpwssd zmm {k}, zmm+3, m128
CPUID Flags: AVX512_4VNNIW
```

Description

Compute 4 sequential operand source-block dot-products of two signed 16-bit element operands with 32-bit element accumulation, and store the results in dst.

Operation

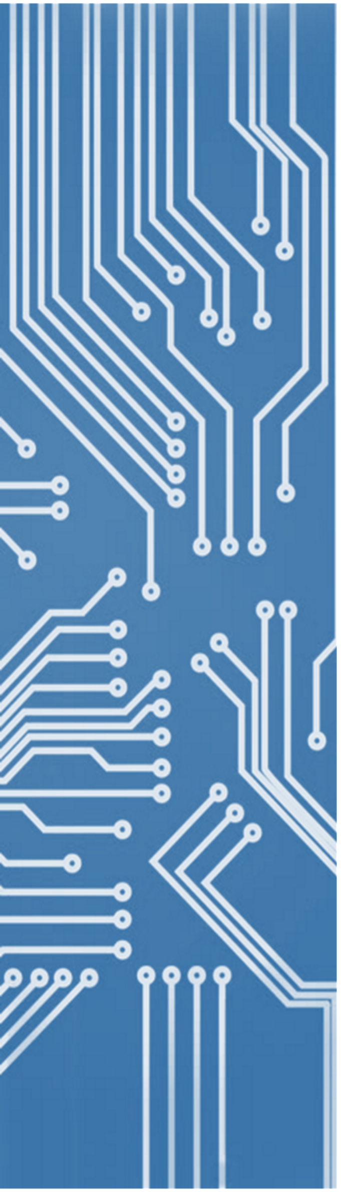
```
FOR j := 0 to 15
        FOR m := 0 to 3
                lim_base := m*32
                i := j*32
                tl := b[lim_base+15:lim_base]
                tu := b[lim_base+31:lim_base+16]
                lword := a{m}[i+15:i] * tl
                uword := a{m}[i+31:i+16] * tu
                dst[i+31:i] := src[i+31:i] + lword + uword
        ENDFOR
ENDFOR
dst[MAX:512] := 0
```

`__m512i _mm512_mask_4dpwssd_epi32 (_m512i src, _mmask16 k, _m512i a0, _m512i a1, _m512i a2, _m512i a3, _m128i * b)` vp4dpwssd

`__m512i _mm512_maskz_4dpwssd_epi32 (_mmask16 k, _m512i src, _m512i a0, _m512i a1, _m512i a2, _m512i a3, _m128i * b)` vp4dpwssd

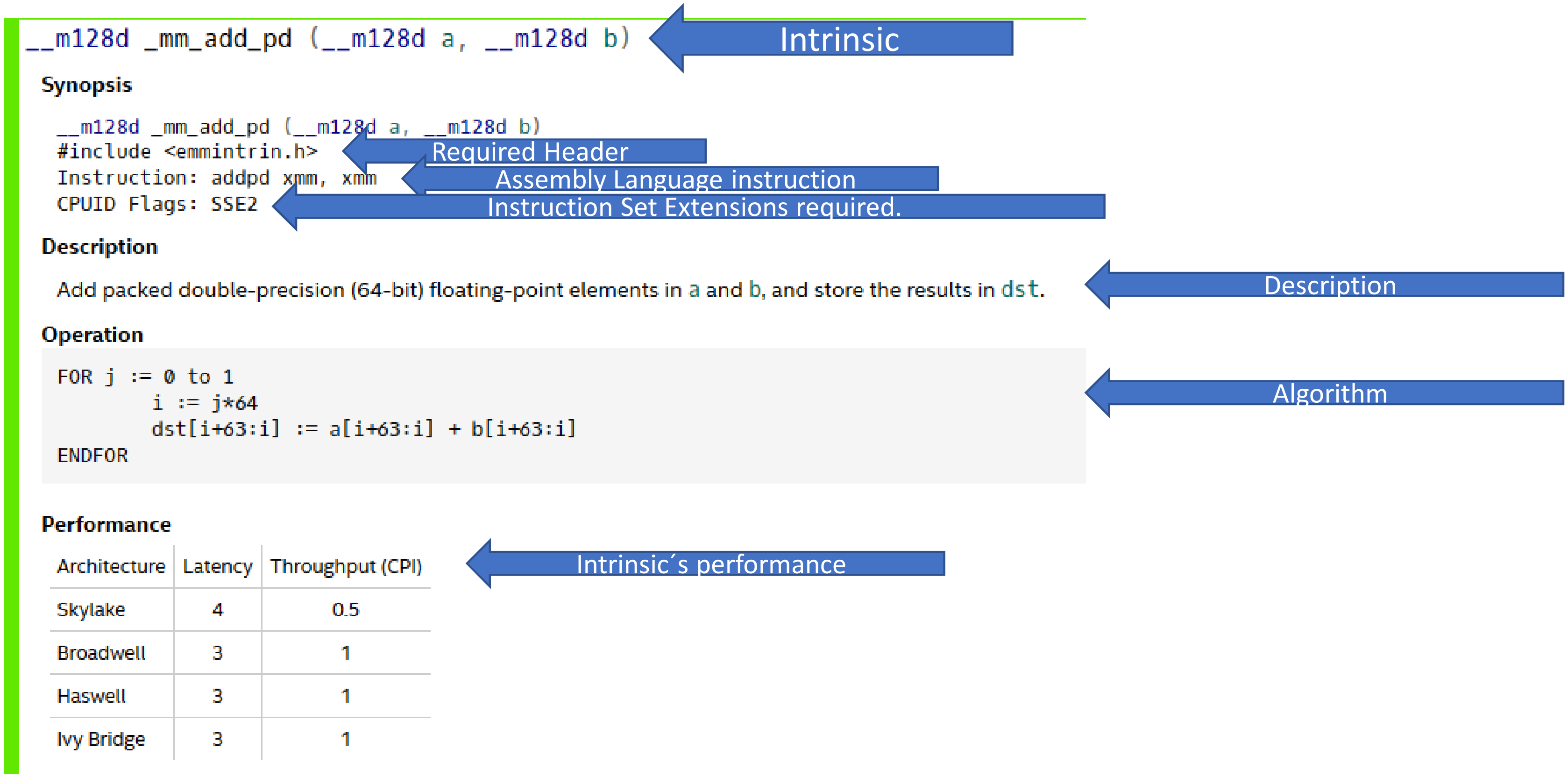
`__m512i _mm512_4dpwssds_epi32 (_m512i src, _m512i a0, _m512i a1, _m512i` vp4dpwssds

https://software.Intel.com/sites/landingpage/IntrinsicsGuide



Intrinsics

Intel Intrinsics Guide



```
__m128d _mm_add_pd (__m128d a, __m128d b)
```

Intrinsic

Synopsis

```
__m128d _mm_add_pd (__m128d a, __m128d b)
#include <emmintrin.h>
Instruction: addpd xmm, xmm
CPUID Flags: SSE2
```

Required Header

Assembly Language instruction

Instruction Set Extensions required.

Description

Add packed double-precision (64-bit) floating-point elements in a and b, and store the results in dst.

Description

Operation

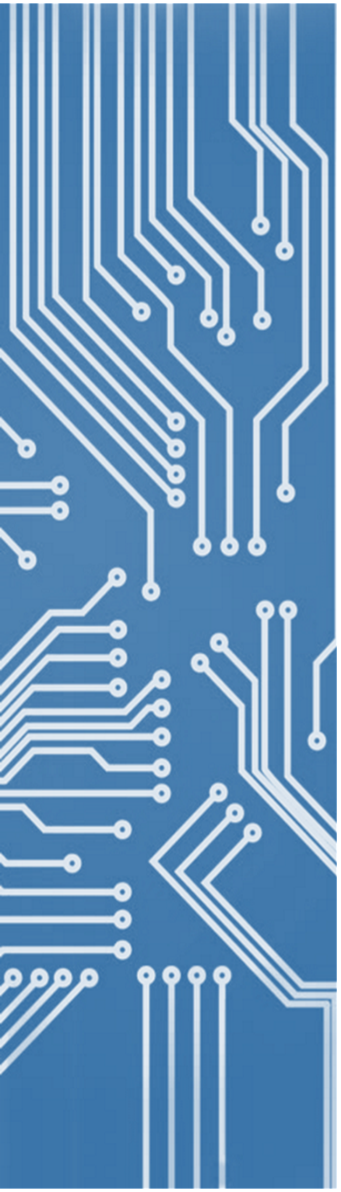
```
FOR j := 0 to 1
	i := j*64
	dst[i+63:i] := a[i+63:i] + b[i+63:i]
ENDFOR
```

Algorithm

Performance

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

Intrinsic´s performance



Intrinsics

- For which datatypes do they work?

Arithmetic int, float and double, logic, and bit-oriented where the same operation may be applied to several data.

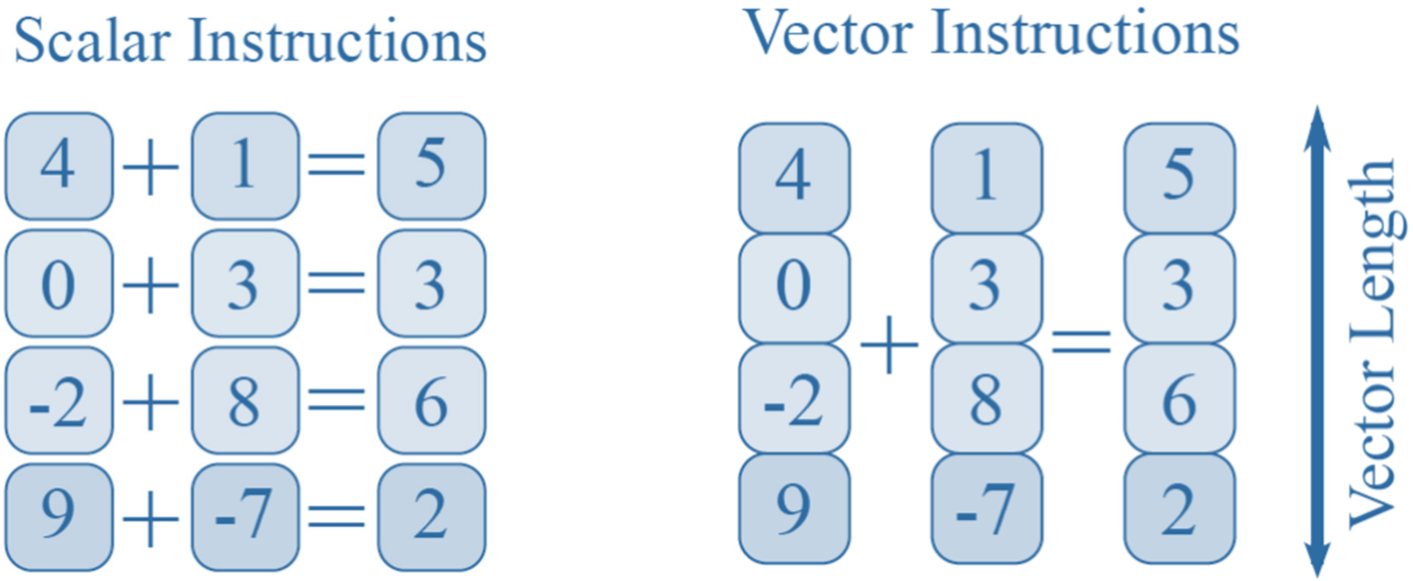
- Is it a standard?

Yes, there are several: MMX, SSE, SSE2, SSE3.X, SSE4.X, AVX, AVX2.

- Is it useful?

Certainly!! (lots of applications require vector operations) and it is feasible (due the large amount of available silicon into the microprocessor).

Remember our example?



Scalar Instructions

4 + 1 = 5

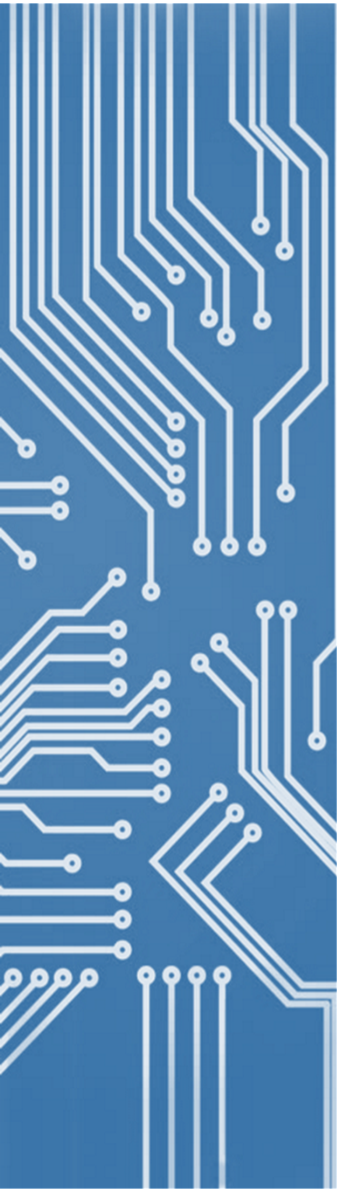
0 + 3 = 3

-2 + 8 = 6

9 + -7 = 2

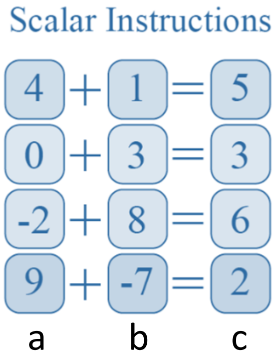
Vector Instructions

$$\begin{bmatrix}4\\0\\-2\\9\end{bmatrix}+\begin{bmatrix}1\\3\\8\\-7\end{bmatrix}=\begin{bmatrix}5\\3\\6\\2\end{bmatrix}$$



Scalar code…

```c
#include <stdio.h>

void main(){
int a[] = {4,0,-2,9};
int b[] = {1,3,8,-7};
int c[] = {0,0,0,0};

int i;
for (i = 0; i <= 3; i++)
c[i] = a[i] + b[i];

for (i = 0; i <= 3; i++)
printf("%d\n", c[i]);

}
```



Scalar Instructions

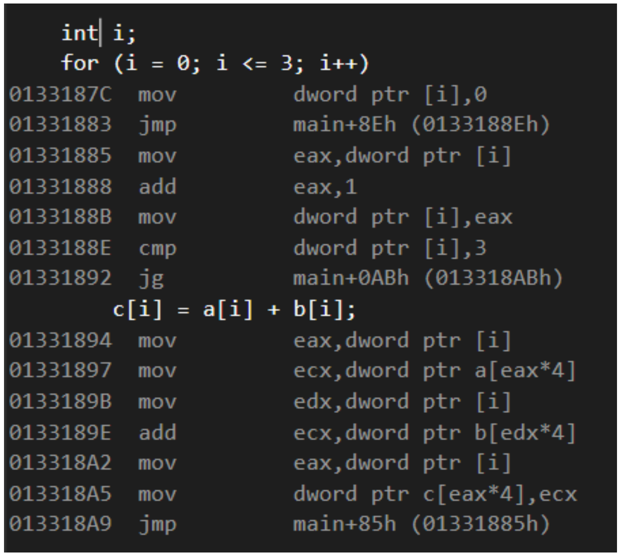
4 + 1 = 5

0 + 3 = 3

-2 + 8 = 6

9 + -7 = 2

a b c

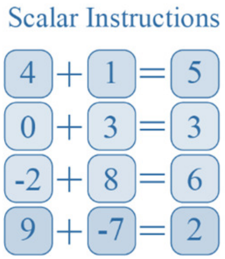


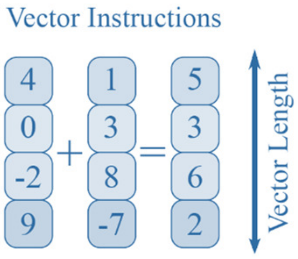
```
    int i;
    for (i = 0; i <= 3; i++)
0133187C  mov          dword ptr [i],0
01331883  jmp          main+8Eh (0133188Eh)
01331885  mov          eax,dword ptr [i]
01331888  add          eax,1
0133188B  mov          dword ptr [i],eax
0133188E  cmp          dword ptr [i],3
01331892  jg           main+0ABh (013318ABh)
        c[i] = a[i] + b[i];
01331894  mov          eax,dword ptr [i]
01331897  mov          ecx,dword ptr a[eax*4]
0133189B  mov          edx,dword ptr [i]
0133189E  add          ecx,dword ptr b[edx*4]
013318A2  mov          eax,dword ptr [i]
013318A5  mov          dword ptr c[eax*4],ecx
013318A9  jmp          main+85h (01331885h)
```

X 4

NOTE THIS: **52 Instructions**

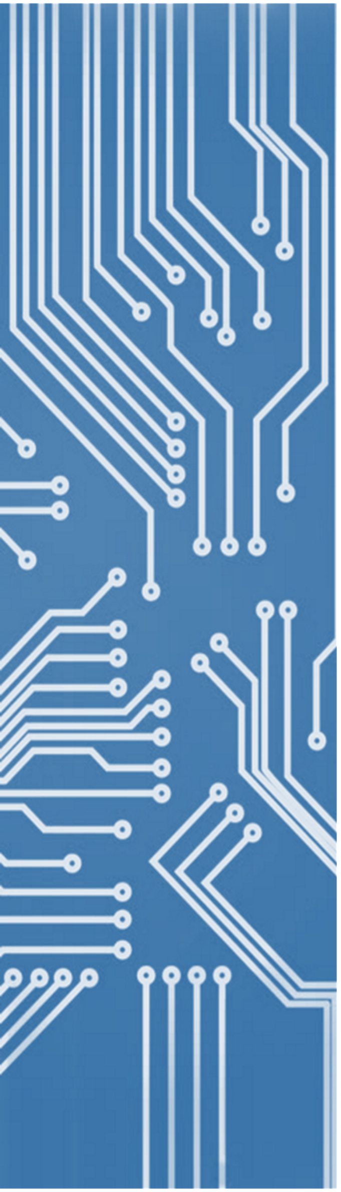
Equivalent VECTOR code with INTRINSICS



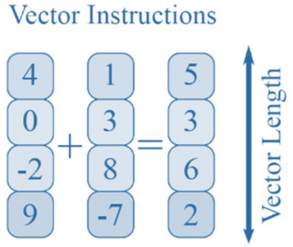


```c
#include <stdio.h>

void main(){
int a[] = {4,0,-2,9};
int b[] = {1,3,8,-7};
int c[] = {0,0,0,0};

int i;
for (i = 0; i <= 3; i++)
c[i] = a[i] + b[i];

for (i = 0; i <= 3; i++)
printf("%d\n", c[i]);
}
```

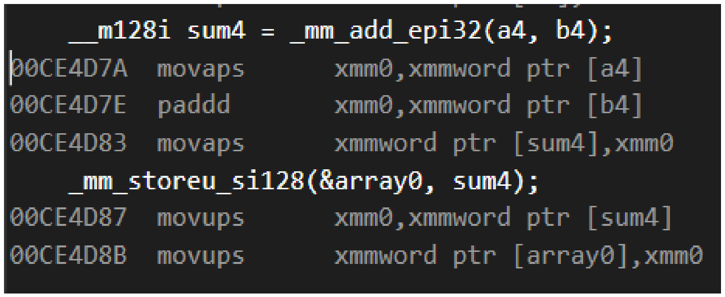
```c
#include <stdio.h>
#include <emmintrin.h>
int main() {
int array0[4];

__m128i a4 = _mm_set_epi32(9,-2,0,4);
__m128i b4 = _mm_set_epi32(-7,8,3,1);
__m128i sum4 = _mm_add_epi32(a4, b4);
_mm_storeu_si128((__m128i *)&array0, sum4);

for (int i = 0; i <= 3; i++)
printf("%d\n", array0[i]);
return 0;
}
```



Equivalent VECTOR code with INTRINSICS



```c
#include <stdio.h>
#include <emmintrin.h>
int main() {
int array0[4];

__m128i a4 = _mm_set_epi32(9,-2,0,4);
__m128i b4 = _mm_set_epi32(-7,8,3,1);
__m128i sum4 = _mm_add_epi32(a4, b4);
_mm_storeu_si128((__m128i *)&array0, sum4);

for (int i = 0; i <= 3; i++)
printf("%d\n", array0[i]);
return 0;
}
```

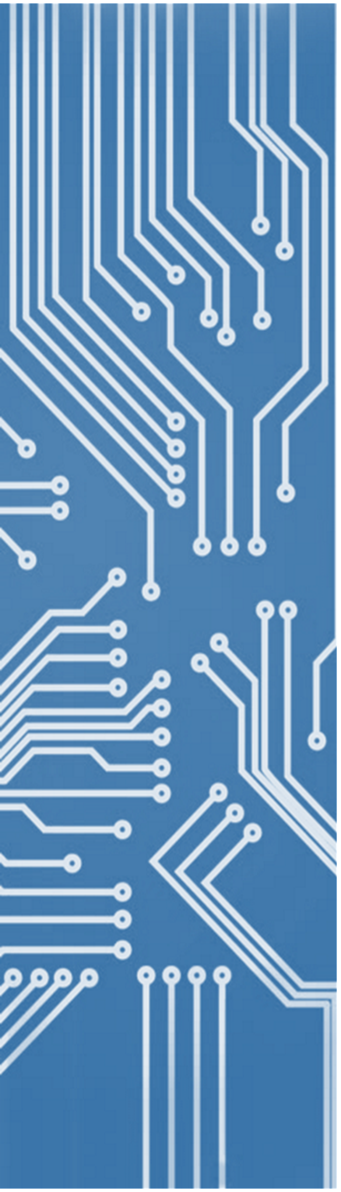


```
    __m128i sum4 = _mm_add_epi32(a4, b4);
00CE4D7A  movaps      xmm0,xmmword ptr [a4]
00CE4D7E  paddd       xmm0,xmmword ptr [b4]
00CE4D83  movaps      xmmword ptr [sum4],xmm0
    _mm_storeu_si128(&array0, sum4);
00CE4D87  movups      xmm0,xmmword ptr [sum4]
00CE4D8B  movups      xmmword ptr [array0],xmm0
```

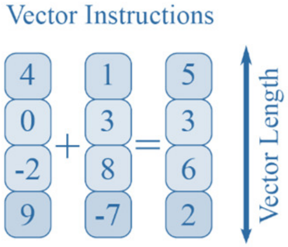
NOTE THIS: 5 Instructions!!!!

Vs.

Scalar Implementation: 52 Instructions



GCC and INTRINSICS

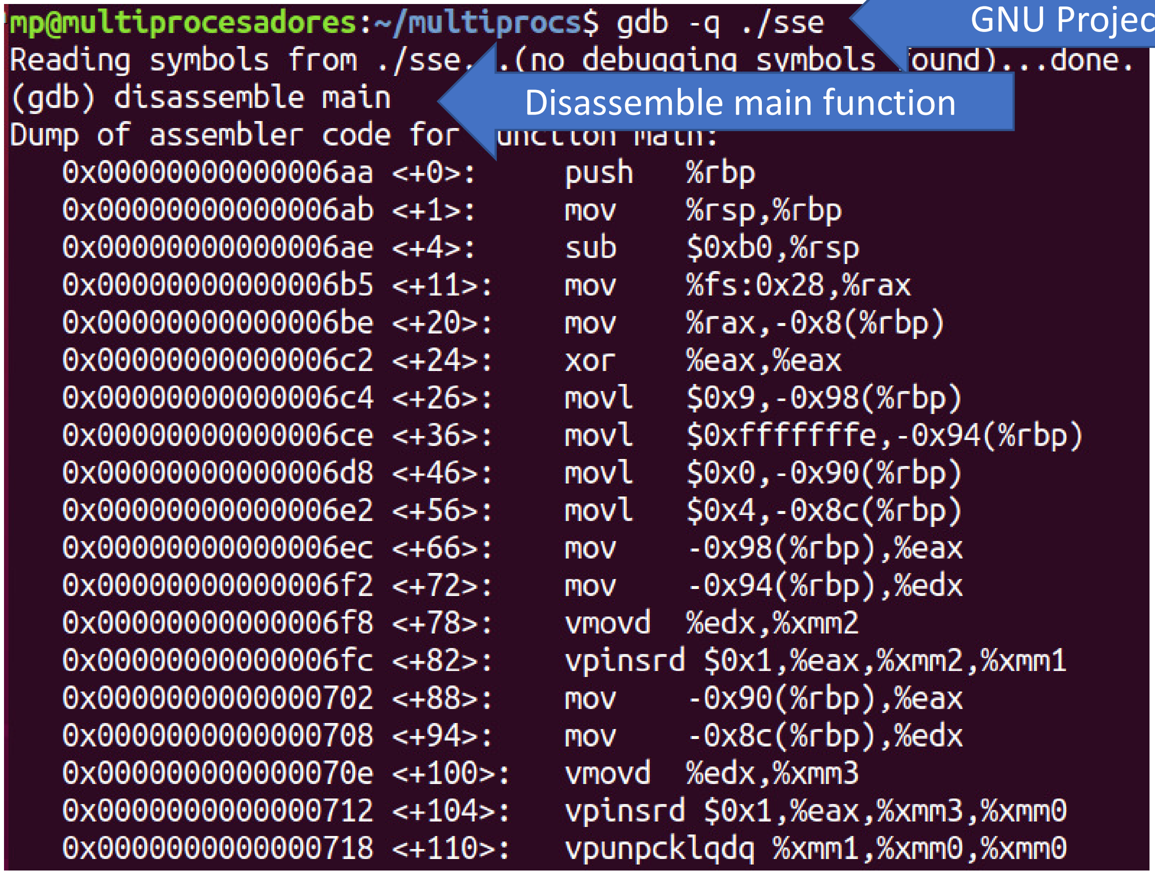


```
#include <stdio.h>
#include <emmintrin.h>
int main() {
int array0[4];

__m128i a4 = _mm_set_epi32(9,-2,0,4);
__m128i b4 = _mm_set_epi32(-7,8,3,1);
__m128i sum4 = _mm_add_epi32(a4, b4);
_mm_storeu_si128((__m128i *)&array0, sum4);

for (int i = 0; i <= 3; i++)
printf("%d\n", array0[i]);
return 0;
}
```

```
$ gcc file.c -o file
```

```
$ man gcc
```

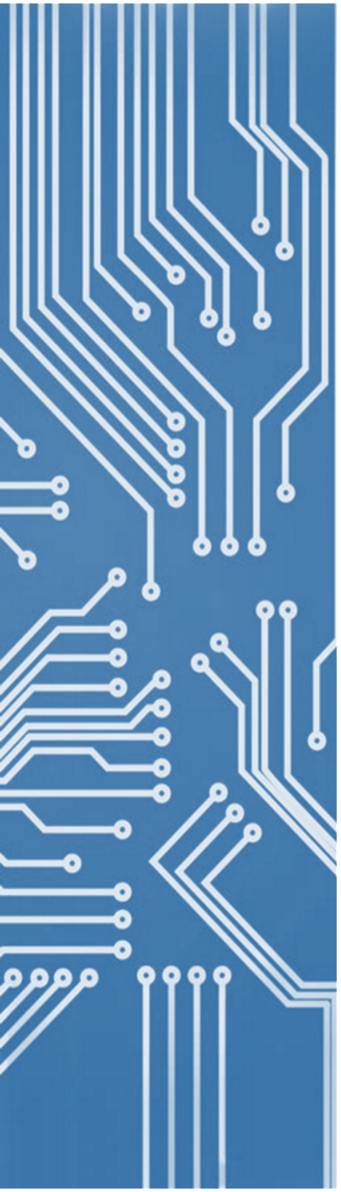
GCC and Disassemble



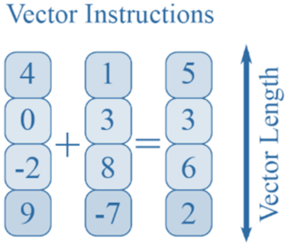
```
mp@multiprocesadores:~/multiprocs$ gdb -q ./sse
Reading symbols from ./sse...(no debugging symbols found)...done.
(gdb) disassemble main
Dump of assembler code for function main:
   0x00000000000006aa <+0>:     push   %rbp
   0x00000000000006ab <+1>:     mov    %rsp,%rbp
   0x00000000000006ae <+4>:     sub    $0xb0,%rsp
   0x00000000000006b5 <+11>:    mov    %fs:0x28,%rax
   0x00000000000006be <+20>:    mov    %rax,-0x8(%rbp)
   0x00000000000006c2 <+24>:    xor    %eax,%eax
   0x00000000000006c4 <+26>:    movl   $0x9,-0x98(%rbp)
   0x00000000000006ce <+36>:    movl   $0xfffffffe,-0x94(%rbp)
   0x00000000000006d8 <+46>:    movl   $0x0,-0x90(%rbp)
   0x00000000000006e2 <+56>:    movl   $0x4,-0x8c(%rbp)
   0x00000000000006ec <+66>:    mov    -0x98(%rbp),%eax
   0x00000000000006f2 <+72>:    mov    -0x94(%rbp),%edx
   0x00000000000006f8 <+78>:    vmovd  %edx,%xmm2
   0x00000000000006fc <+82>:    vpinsrd $0x1,%eax,%xmm2,%xmm1
   0x0000000000000702 <+88>:    mov    -0x90(%rbp),%eax
   0x0000000000000708 <+94>:    mov    -0x8c(%rbp),%edx
   0x000000000000070e <+100>:   vmovd  %edx,%xmm3
   0x0000000000000712 <+104>:   vpinsrd $0x1,%eax,%xmm3,%xmm0
   0x0000000000000718 <+110>:   vpunpcklqdq %xmm1,%xmm0,%xmm0
```

GNU Project Debugger

Disassemble main function



Important Notes: Memory arrangement…

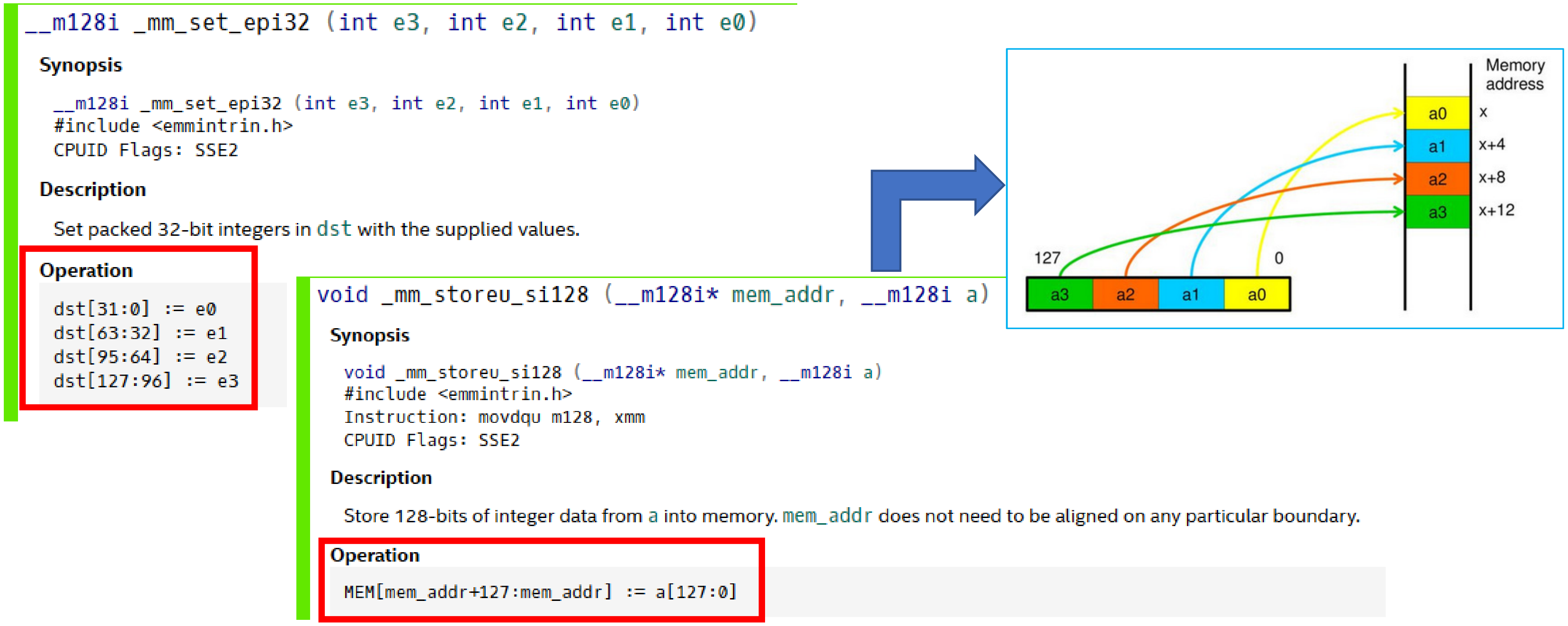


```
int a[] = {4,0,-2,9};
```

a[0], a[1], a[2], a[3]

```
__m128i a4 = _mm_set_epi32(9,-2,0,4);
```

a[3], a[2], a[1], a[0]



```
__m128i _mm_set_epi32 (int e3, int e2, int e1, int e0)
```

Synopsis

```
__m128i _mm_set_epi32 (int e3, int e2, int e1, int e0)
#include <emmintrin.h>
CPUID Flags: SSE2
```

Description

Set packed 32-bit integers in dst with the supplied values.

Operation

```
dst[31:0] := e0
dst[63:32] := e1
dst[95:64] := e2
dst[127:96] := e3
```

```
void _mm_storeu_si128 (__m128i* mem_addr, __m128i a)
```

Synopsis

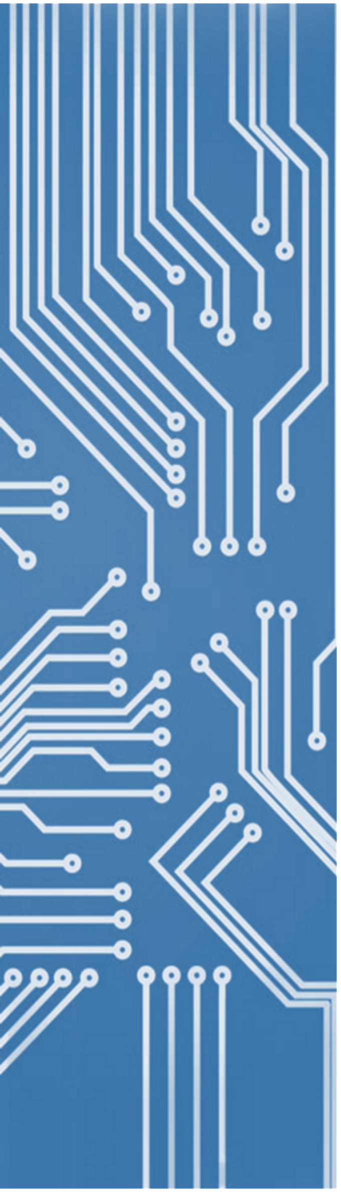
```
void _mm_storeu_si128 (__m128i* mem_addr, __m128i a)
#include <emmintrin.h>
Instruction: movdqu m128, xmm
CPUID Flags: SSE2
```

Description

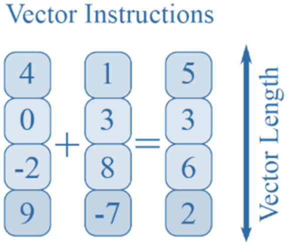
Store 128-bits of integer data from a into memory. mem_addr does not need to be aligned on any particular boundary.

Operation

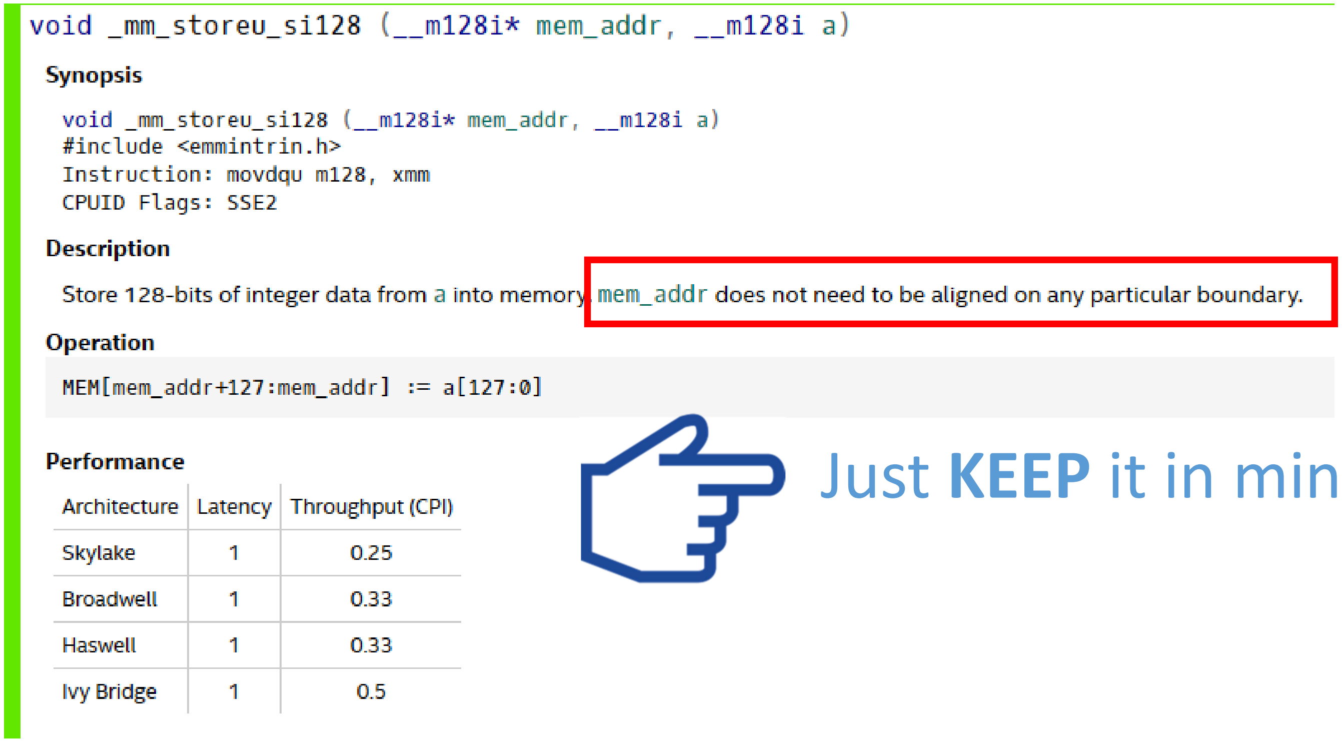
```
MEM[mem_addr+127:mem_addr] := a[127:0]
```



Important Notes: Memory alignment…



```
_mm_storeu_si128(&array0, sum4);
```



```
void _mm_storeu_si128 (__m128i* mem_addr, __m128i a)
```

Synopsis

```
void _mm_storeu_si128 (__m128i* mem_addr, __m128i a)
#include <emmintrin.h>
Instruction: movdqu m128, xmm
CPUID Flags: SSE2
```

Description

Store 128-bits of integer data from a into memory. mem_addr does not need to be aligned on any particular boundary.

Operation

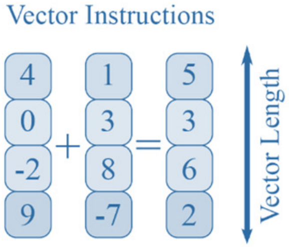
```
MEM[mem_addr+127:mem_addr] := a[127:0]
```

Performance

Architecture	Latency	Throughput (CPI)
Skylake	1	0.25
Broadwell	1	0.33
Haswell	1	0.33
Ivy Bridge	1	0.5

Just **KEEP** it in mind for now…

Lets analyze the code…



```c
#include <stdio.h>
#include <emmintrin.h>

int main() {
int array0[4];

__m128i a4 = _mm_set_epi32(4, 0, -2, 9);
__m128i b4 = _mm_set_epi32(1, 3, 8, -7);
__m128i sum4 = _mm_add_epi32(a4, b4);
_mm_storeu_si128((__m128i *)&array0, sum4);

for (int i = 0; i <= 3; i++)
printf("%d\n", array0[i]);

return 0;
}
```

SSE2 Intrinsics Header File

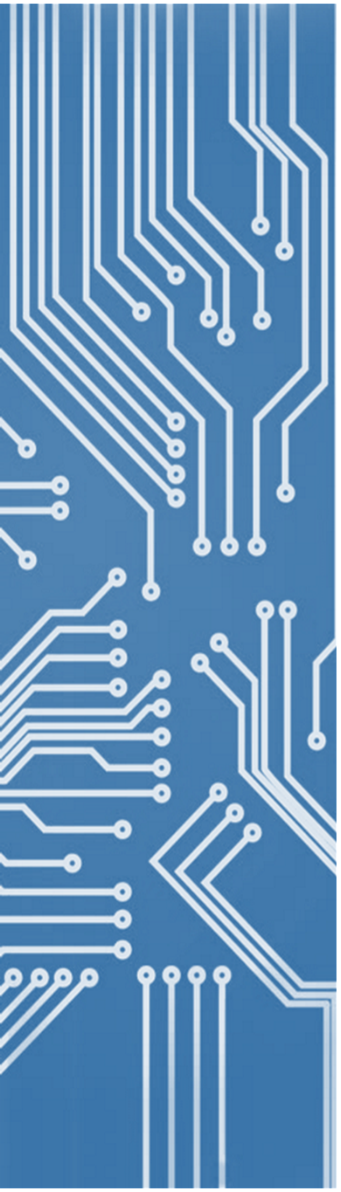
Data Type: 128 bit wide, INTEGER

Set packed 32-bit integers in dst with the supplied values.

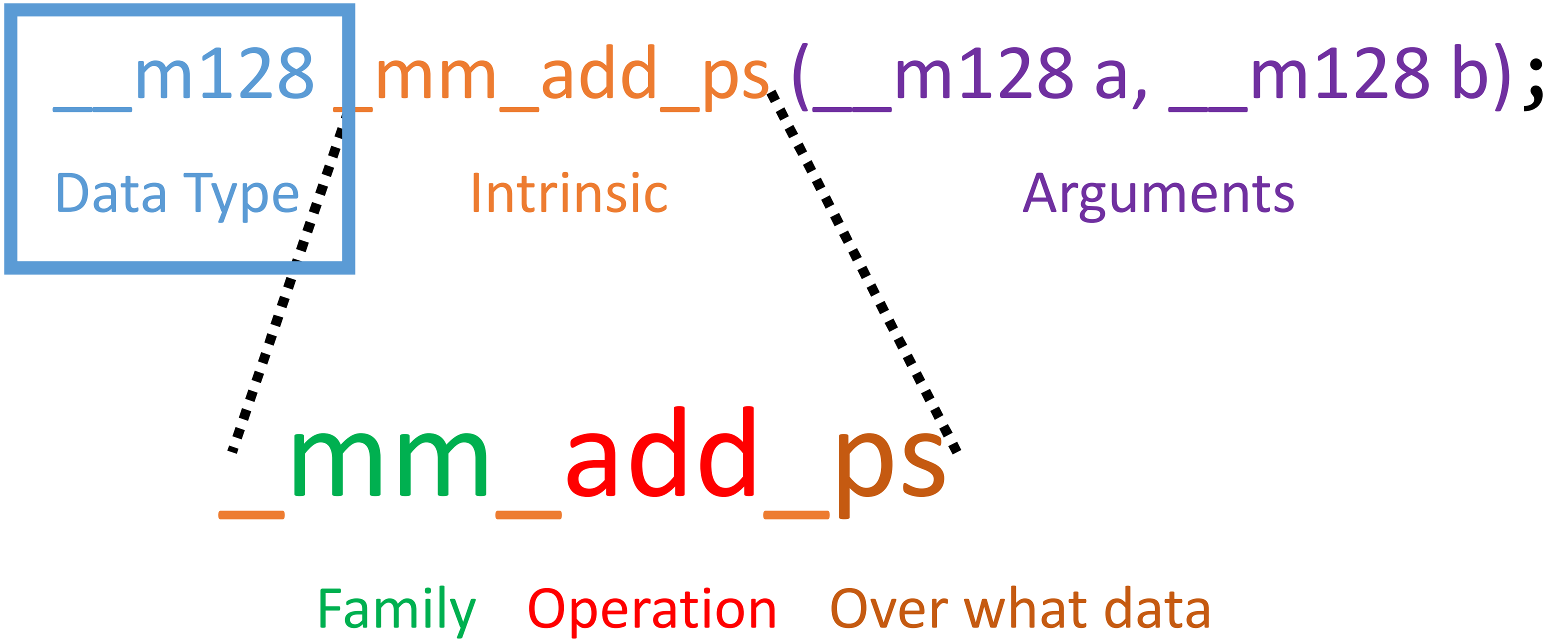
Add packed 32-bit integers in a and b, store the results in dst.

Store 128-bits of integer data from variable into memory.

IMPORTANT NOTE: Variables of type **__m128i** are automatically aligned on 16-byte boundaries.



Lets analyze the instructions…



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

Data Type

Intrinsic

Arguments

```
_mm_add_ps
```

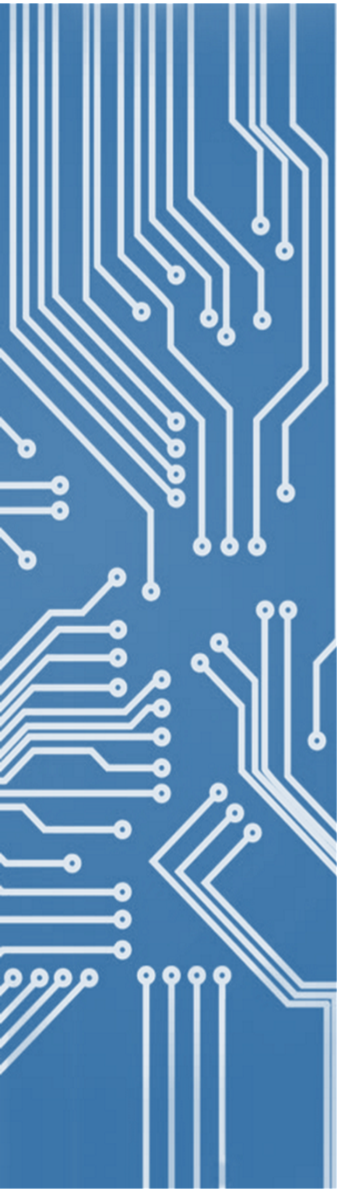
Family

Operation

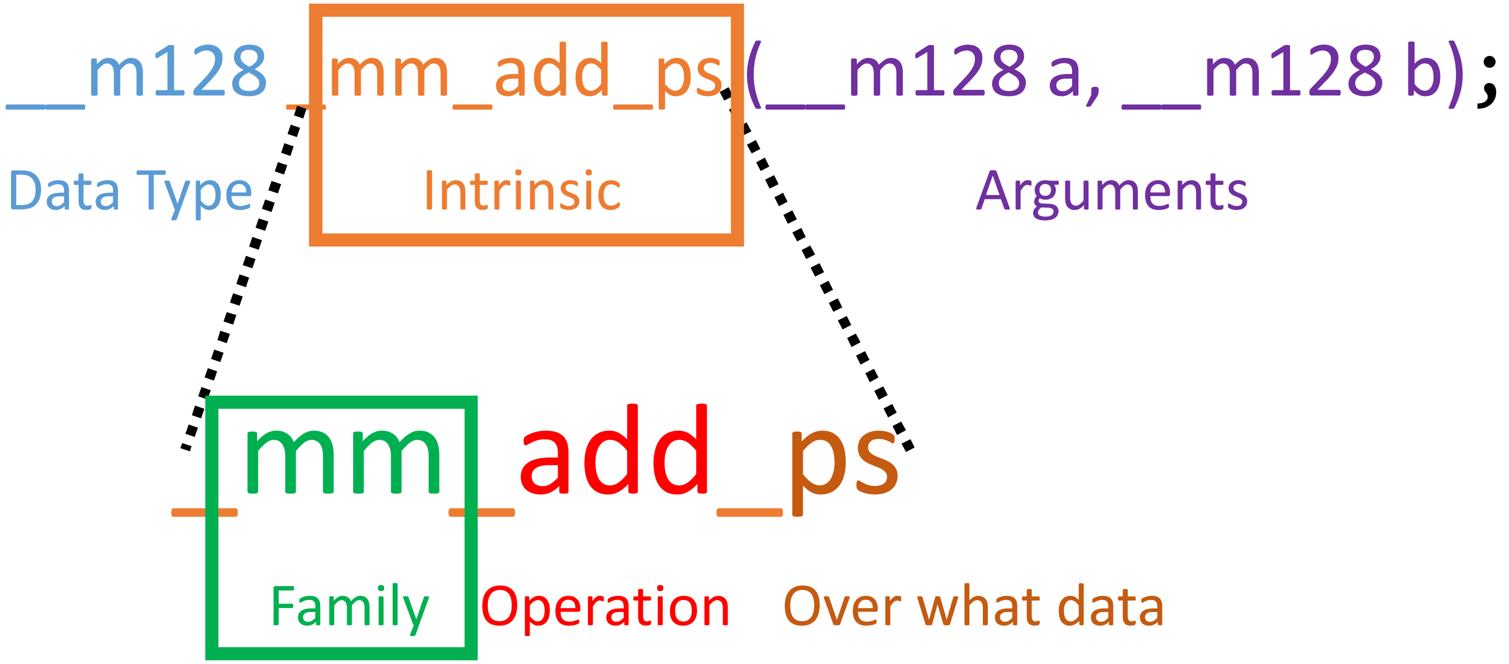
Over what data

Data Types for Intrinsics

Data Type	Available from	Meaning
__m64	MMX	The __m64 data type is used to represent the contents of an MMX™ register. The __m64 data type can hold eight 8-bit values, four 16-bit values, two 32-bit values, or one 64-bit value.
__m128	SSE	128-bit single precision floating point (32-bit each)
__m128d	SSE2	128-bit double precision floating point (64-bit each)
__m128i	SSE2	128-bit integers (bytes, words, double words, etc.)
__m256	AVX	256-bit single precision floating point (32-bit each)
__m256d	AVX	256-bit double precision floating point (64-bit each)
__m256i	AVX	256-bit integers (bytes, words, double words, etc.)
__m512	AVX-512	512-bit single precision floating point (32-bit each)
__m512d	AVX-512	512-bit double precision floating point (64-bit each)
__m512i	AVX-512	512-bit integers (bytes, words, double words, etc.)



Lets analyze the instructions…



__m128 _mm_add_ps(__m128 a, __m128 b);

Data Type

Intrinsic

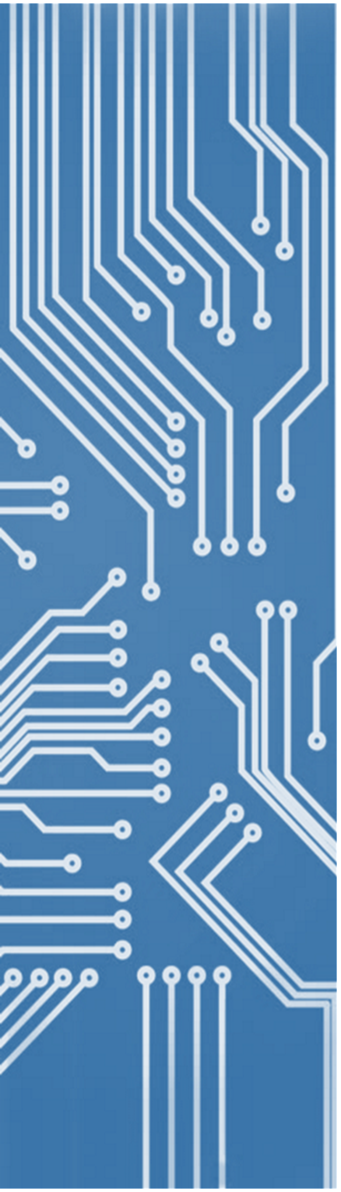
Arguments

_mm_add_ps

Family

Operation

Over what data

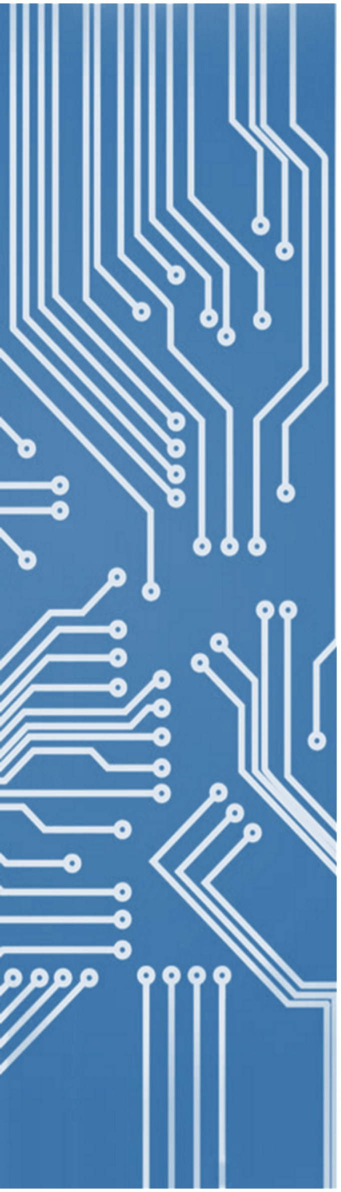


Notation according to Instruction Set Family

	Family
_mm_operation_suffix	SSEx & MMX (128-bit & 64-bit operation)
_mm256_operation_suffix	AVX-AVX2 (256-bit operation)
_mm512_operation_suffix	AVX-512 (512-bit operation)

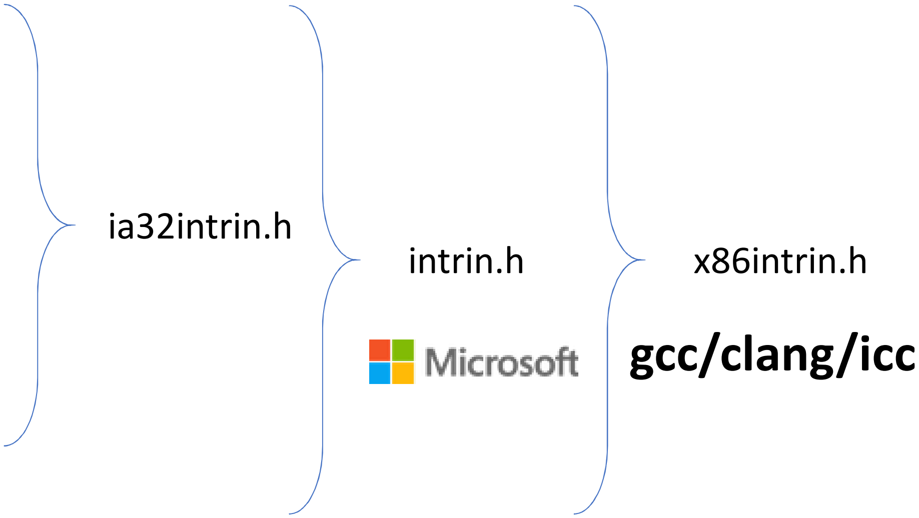
Example of the same operation in 3 different families:

```
__m128i _mm_abs_epi32 (__m128i a)
__m256i _mm256_abs_epi32 (__m256i a)
__m512i _mm512_abs_epi64 (__m512i a)
```

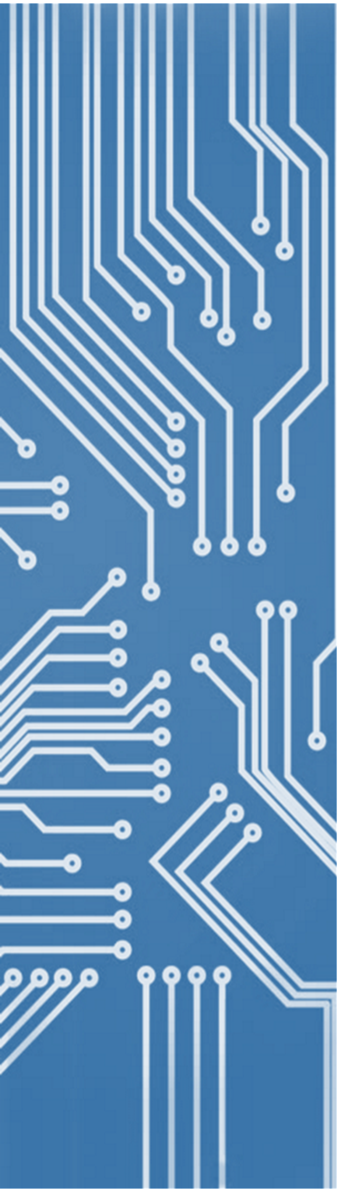


C Header Files according to family.

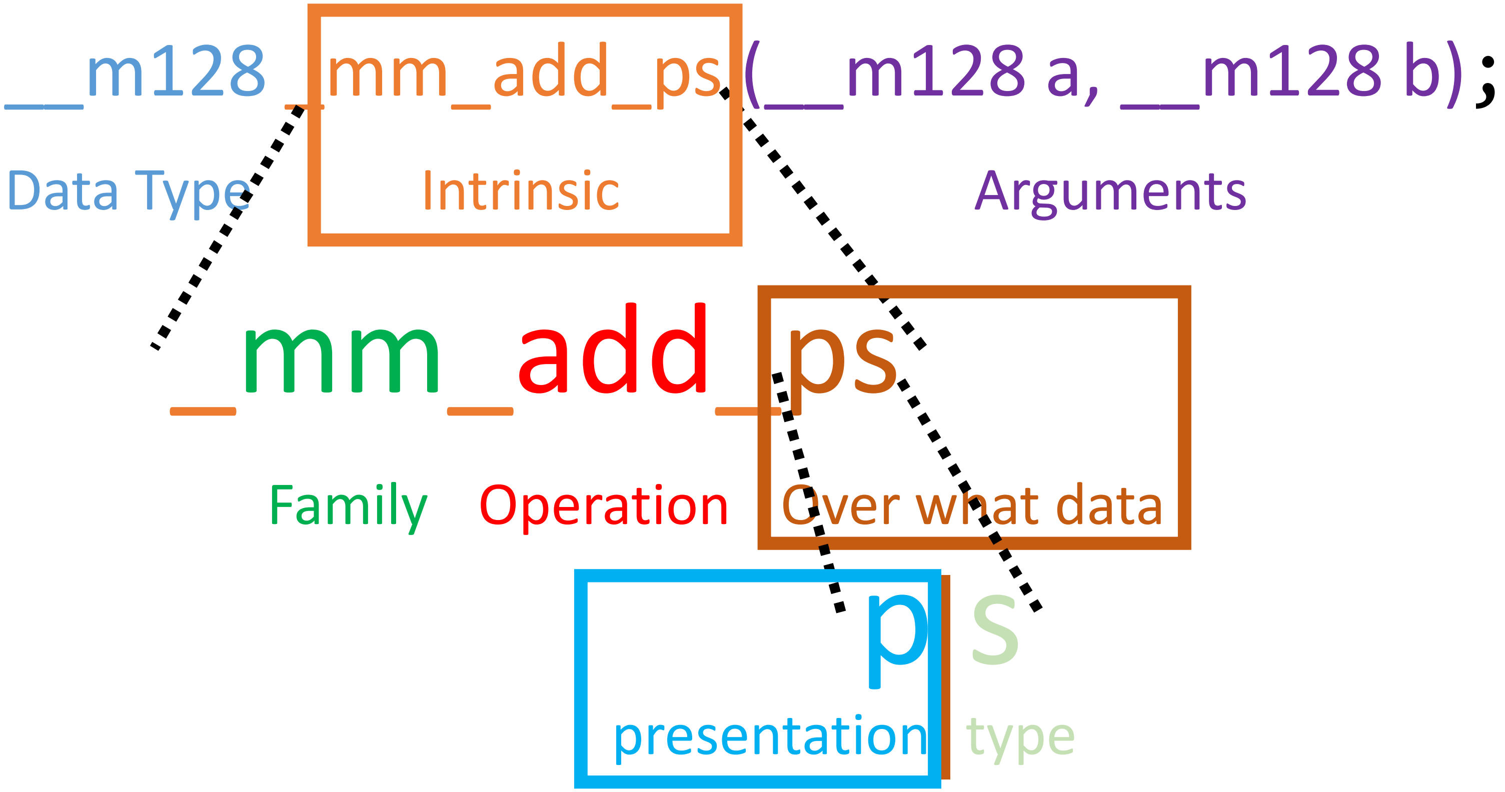
Family	Header File
MMX	mmintrin.h
SSE	xmmintrin.h
SSE2	emmintrin.h
SSE3	pmmintrin.h
SSSE3	tmmintrin.h
SSE4.1	smmintrin.h
SSE4.2	nmmintrin.h
AVX*	immintrin.h



*AVX, AVX2, AVX512, all SSE+MMX (except SSE4A and XOP), popcnt, BMI/BMI2, FMA



Lets analyze the instructions…



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

Data Type

Intrinsic

Arguments

_mm_add_ps

Family

Operation

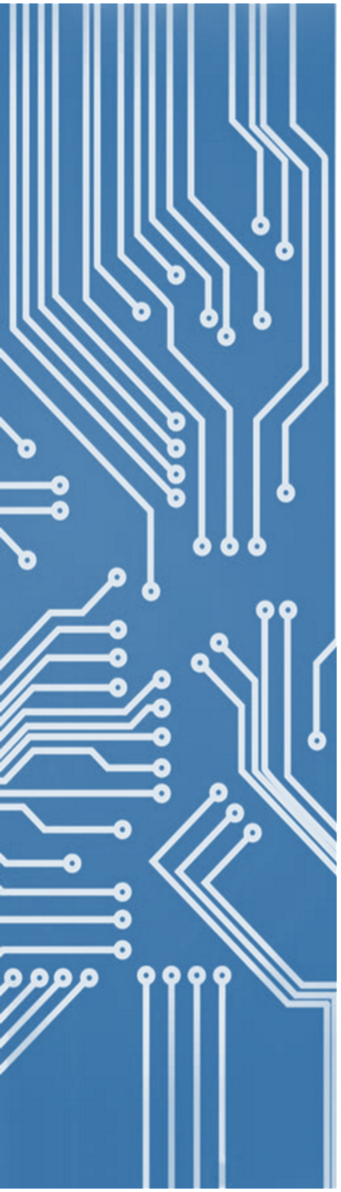
Over what data

p

s

presentation

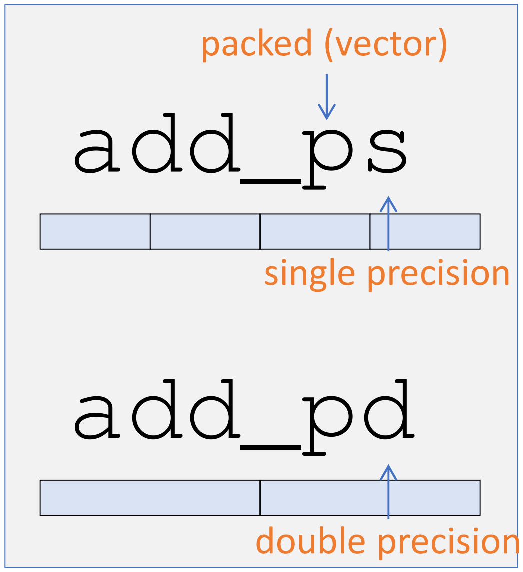
type



Suffix… Presentation

Presentation can be:

- s – Single Slot or Scalar
- p – Packed or Vector
- ep – Extended Packed



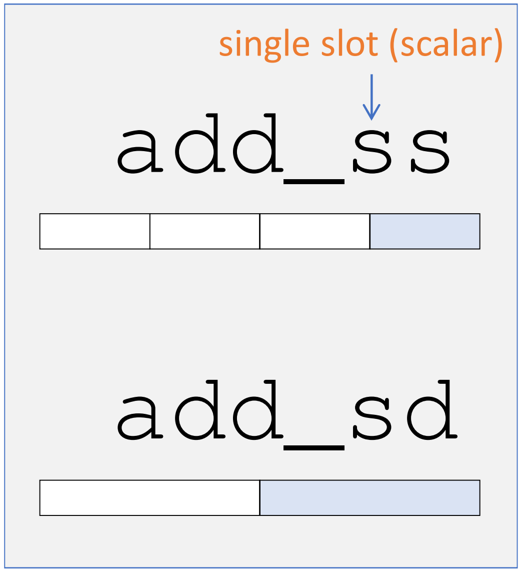
packed (vector)

add_ps

single precision

add_pd

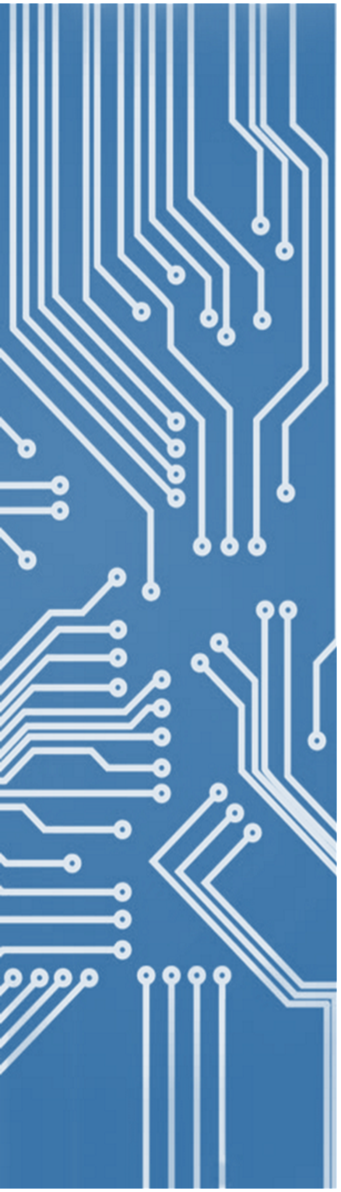
double precision



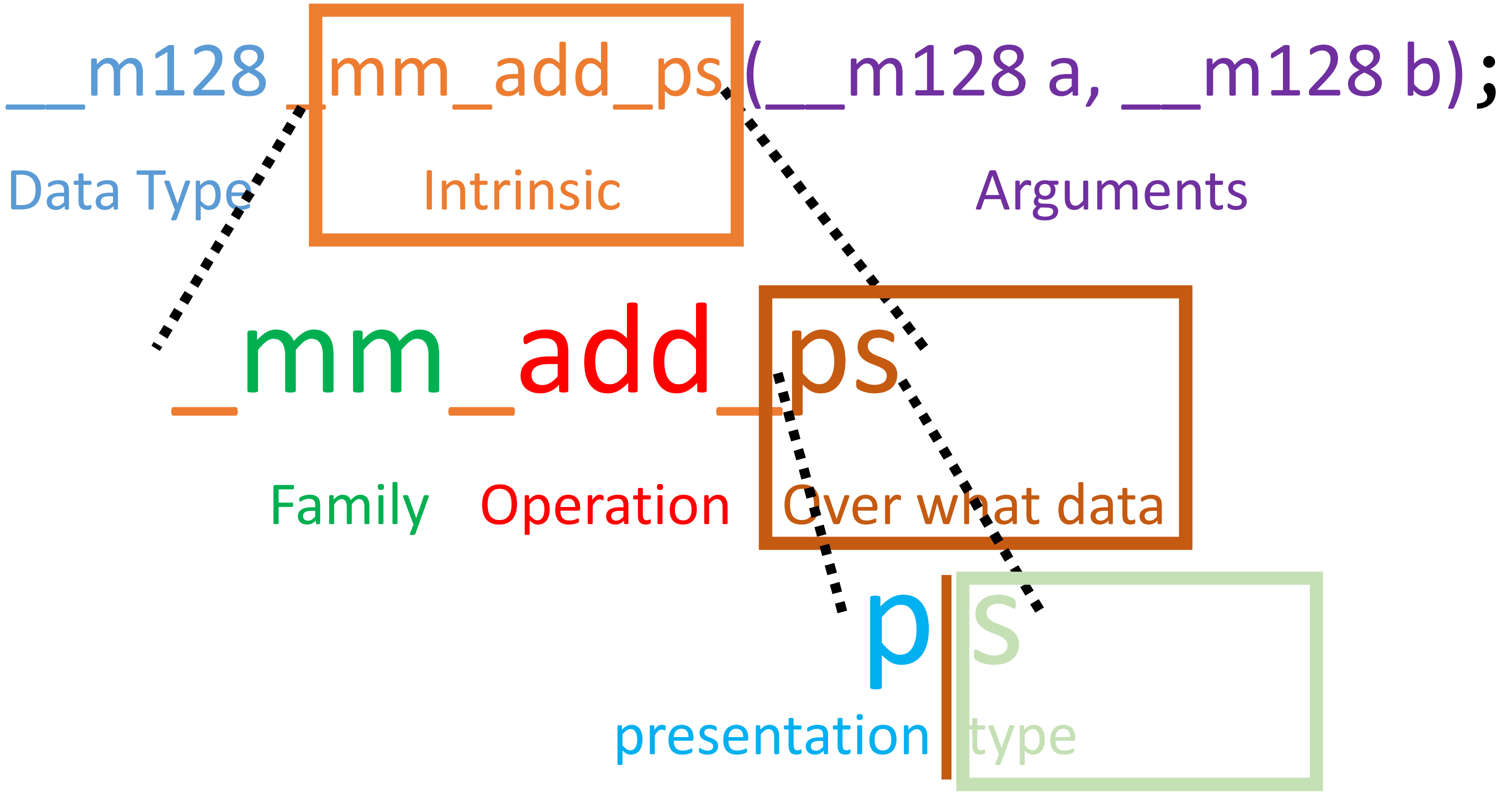
single slot (scalar)

add_ss

add_sd



Lets analyze the instructions…



__m128 _mm_add_ps(__m128 a, __m128 b);

Data Type

Intrinsic

Arguments

_mm_add_ps

Family

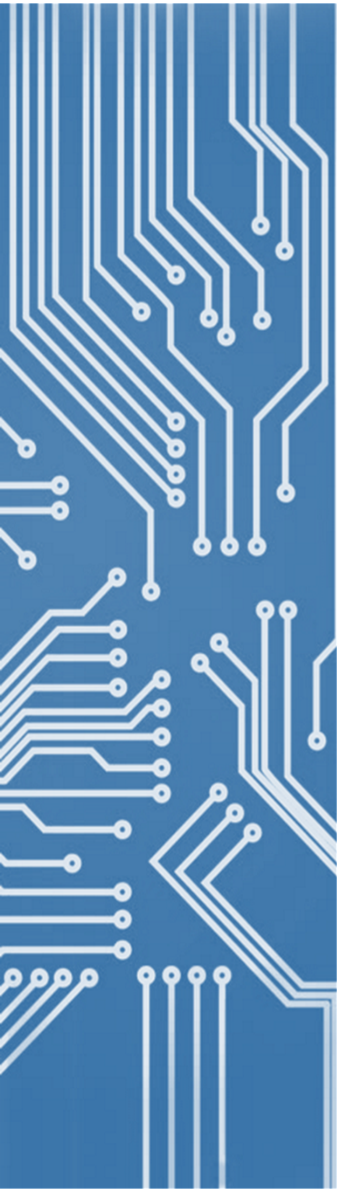
Operation

Over what data

p|s

presentation

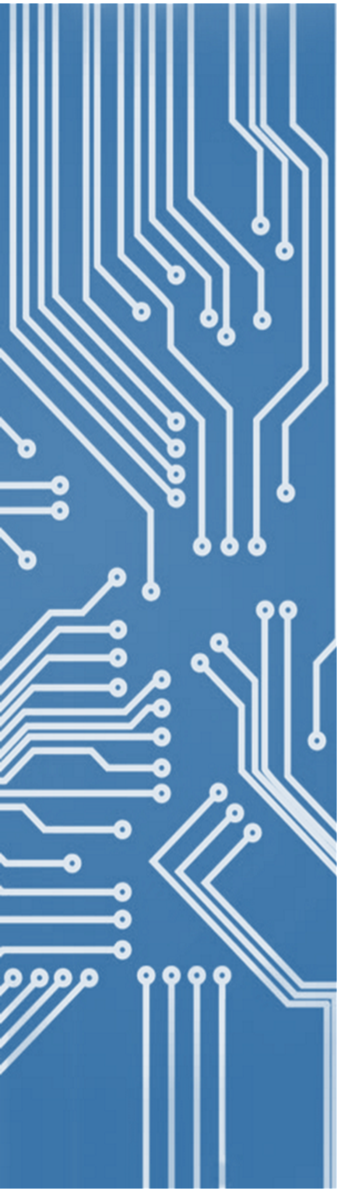
type



Suffix… Type.

Type can be:

- s - single-precision floating point
- d - double-precision floating point
- i8/16/32/64/128 - 8/16/32/64/128-bit signed integer
- u8/16/32/64 - 8/16/32/64-bit unsigned integer



Teams

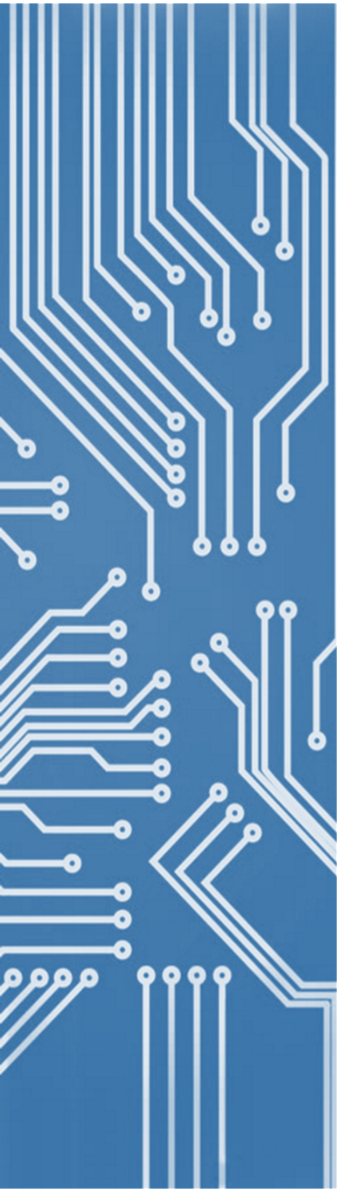
1

Remember the code of our example?

```
#include <stdio.h>
#include <emmintrin.h>
int main() {
int array0[4];

__m128i a4 = _mm_set_epi32(9, -2, 0, 4);
__m128i b4 = _mm_set_epi32(-7, 8, 3, 1);
__m128i sum4 = _mm_add_epi32(a4, b4);
_mm_storeu_si128((__m128i *)&array0, sum4);

for (int i = 0; i <= 3; i++)
printf("%d\n", array0[i]);
return 0;
}
```

Change the necessary lines to be able to do the following sum in just 1 operation:

44, 0, -212, 65, 86, 51, 65, 75
+
31, 4, 220, -60, -86, 4, 35, -85



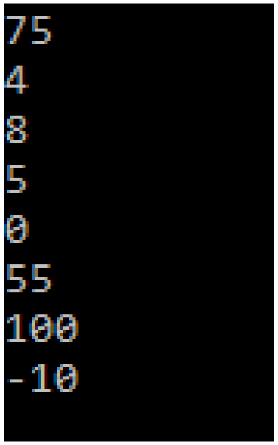
Teams

1

Answer:

```
#include <stdio.h>
#include <immintrin.h>
int main() {
int array0[8];

__m256i a4 = _mm256_set_epi32(75,65,51,86,65,-212,0,44);
__m256i b4 = _mm256_set_epi32(-85,35,4,-86,-60,220,4,31);
__m256i sum4 = _mm256_add_epi32(a4, b4);
_mm256_storeu_si256((__m256i *)&array0, sum4);

for (int i = 0; i <= 7; i++)
printf("%d\n", array0[i]);
return 0;
}
```

```
   44, 0, -212, 65, 86, 51, 65, 75
+
   31, 4, 220, -60, -86, 4, 35, -85
-----------------------------------
   75, 4,    8,    5,  0,55,100,-10
```



```
75
4
8
5
0
55
100
-10
```

Thank you!