



# ***Tema 2: PARALELISMO DE DATOS***

## **Parte 1: Procesadores Vectoriales**

Universidad de Cádiz

# ÍNDICE

## Extensiones SIMD

- Introducción
- Evolución
- Registros XMM, YMM y ZMM
- ¿Como funcionan?
- Ejemplos de código
- Operaciones no destructivas
- ¿Cómo se usan?

# Para saber más

Intel

<http://www.tommesani.com/index.php/simd/46-sse-arithmetic.html>

<http://web.stanford.edu/class/ee382/MISC/amd3dnow.pdf>

<https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions>

<https://software.intel.com/es-es/isa-extensions/intel-avx>

<http://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX>

# Evolución

- ❖ **MMX** (Extension Multimedia) 1997 Intel Pentium
- ❖ **3DNow!** 1998 por AMD, en desuso
- ❖ **SSE** 1999, respuesta de Intel a 3DNow  
(**S**treaming **S**IMD **E**xtension)  
Evoluciones:  
SSE2 (2002), SSE3(2005), SSE4 (2007)
- ❖ **AVX** (**A**dvanced **V**ector **E**xtensions)  
propuesto por Intel en 2008 implementado en 2011  
Evoluciones:  
**AVX2**(2013), **AVX512**(Propuesto 2013)

# MMX

Orientado al uso de programas multimedia

Insertan en las CPU nuevas instrucciones MMX y renombran los registros FPU a MMx

“MMX es una extensión del conjunto de instrucciones existentes (IA32). Hay 57 nuevas instrucciones que los procesadores compatibles con MMX comprenden, y que necesitan nuevos programas para ser explotados.”

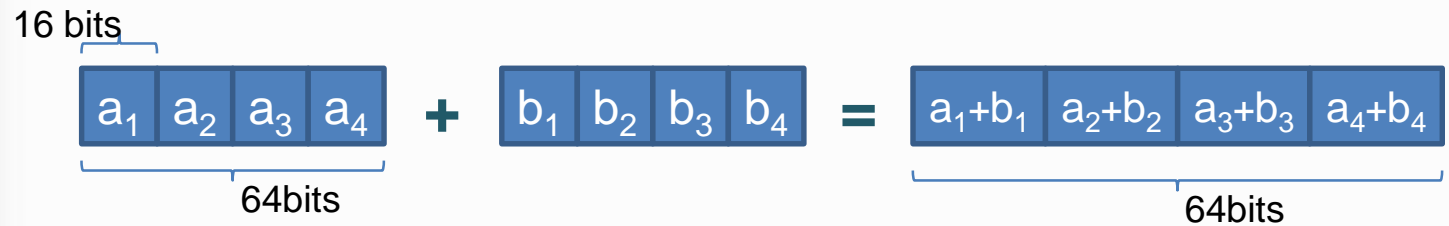


Mayores Inconvenientes:

- FPU y MMX comparten registros así que hay que elegir entre usar uno u otro.
- Hay que programar 2 versiones para los que tienen MMX y para los que no.

# MMX

Permite hasta 4 operaciones simultaneas descomponiendo los registros



Solo trabaja con números enteros

# MMX

Ejemplo...¿Qué hace este programa? ¿Qué mejora?

```
char d[] = {5,5,5,5,5,5,5,5}           // 8 bytes
char clr = {12,45,15,34,76,13,...23,65} //24 byte

__asm {
    movq mm1, d           // load constant in mm1
    mov cx, 3             // initialize loop counter
    mov esi, 0            // set index to 0
L1: movq mm0, clr[esi]    // load 8 bytes of vector into mm0
    paddb mm0, mm1        // performed vector add op
    movq clr[esi], mm0    // store result
    add esi, 8            // update index
    loop L1
    emms                  // Clear mmx register state
}
```

# 3DNow!

- ❖ Extensión multimedia creada por AMD.
- ❖ Usa los registros MMX de 64 bits
- ❖ Inserta 21 instrucciones que permiten cálculos en coma flotante de precisión simple
- ❖ Orientado al procesamiento de vectores y juegos 3D.
- ❖ La escasez de registros obliga a usar demasiado la memoria principal para pasos intermedios.
- ❖ La poca implementación frente al uso masivo de SSE hace que se declare obsoleta en 2010





# SSE

- ❖ Añade registros independientes de MMX  
8 registros de 128 bits (XMM0...XMM7)
- ❖ Añade registro de estado específico (MXCSR)
- ❖ Añade soporte para operaciones de vectores en coma flotante (*Single precision*)
- ❖ Añade 70 instrucciones multimedia  
Algunas son instrucciones específicas para mejorar la cache L2-RAM
- ❖ Diseñadas para la ejecución de un flujo constante (como reproducción de video/audio)

# Evolución SSE (SSE2, SSE3, SSE4)

- ❖ La ultima actualización de SSE es la 4
- ❖ En cada evolución se incorporan nuevas instrucciones permitiendo operar más rápidamente
- ❖ Entre otras se introducen instrucciones específicas de:
  - Video
  - Encriptación
  - Reconocimiento de voz
  - Procesado de fotos
- ❖ Obliga a reescribir los programas para aprovechar la tecnología

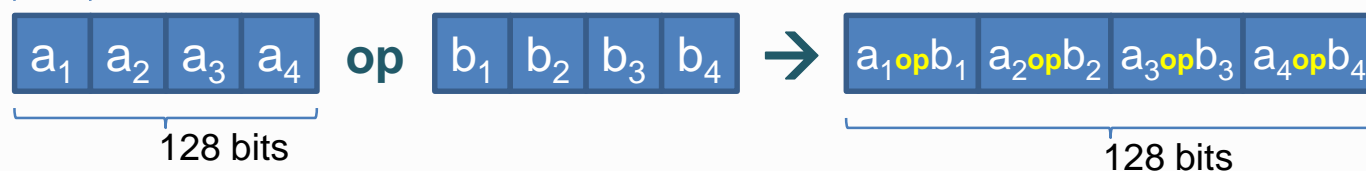
# Evolución SSE (SSE2, SSE3, SSE4)

- ❖ Incorporan instrucciones de operaciones en coma flotante de 64 bits (Double precision) (a partir de SSE2)
- ❖ Permite operaciones vectoriales de enteros dividiendo los registros de 128 en
  - 2 de 64 bits (*quad word*)    8 de 16 bits (*words*)
  - 4 de 32 bits (*double word*)    16 de 8 bits (*single bytes*)
- ❖ Incorporan la operaciones horizontales (a partir de SSE3)
- ❖ Incorpora H.264 *video encoding* (a partir de SSE3)
- ❖ Incorporan instrucciones de tratamiento de texto (a partir de SSE4.2)

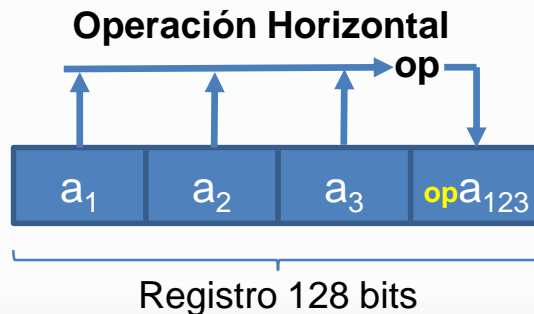
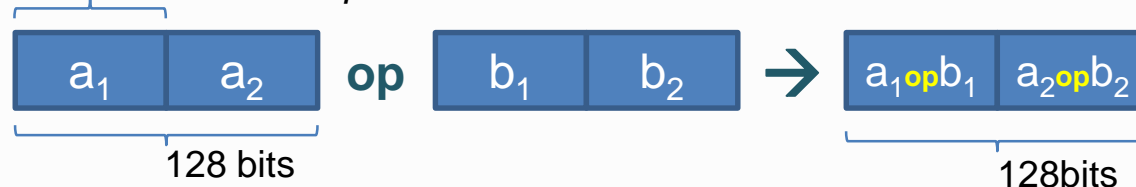
# Evolución SSE (SSE2, SSE3, SSE4)

## ❖ Ejemplos de operaciones admitidas

32 bits enteros o *single precision*



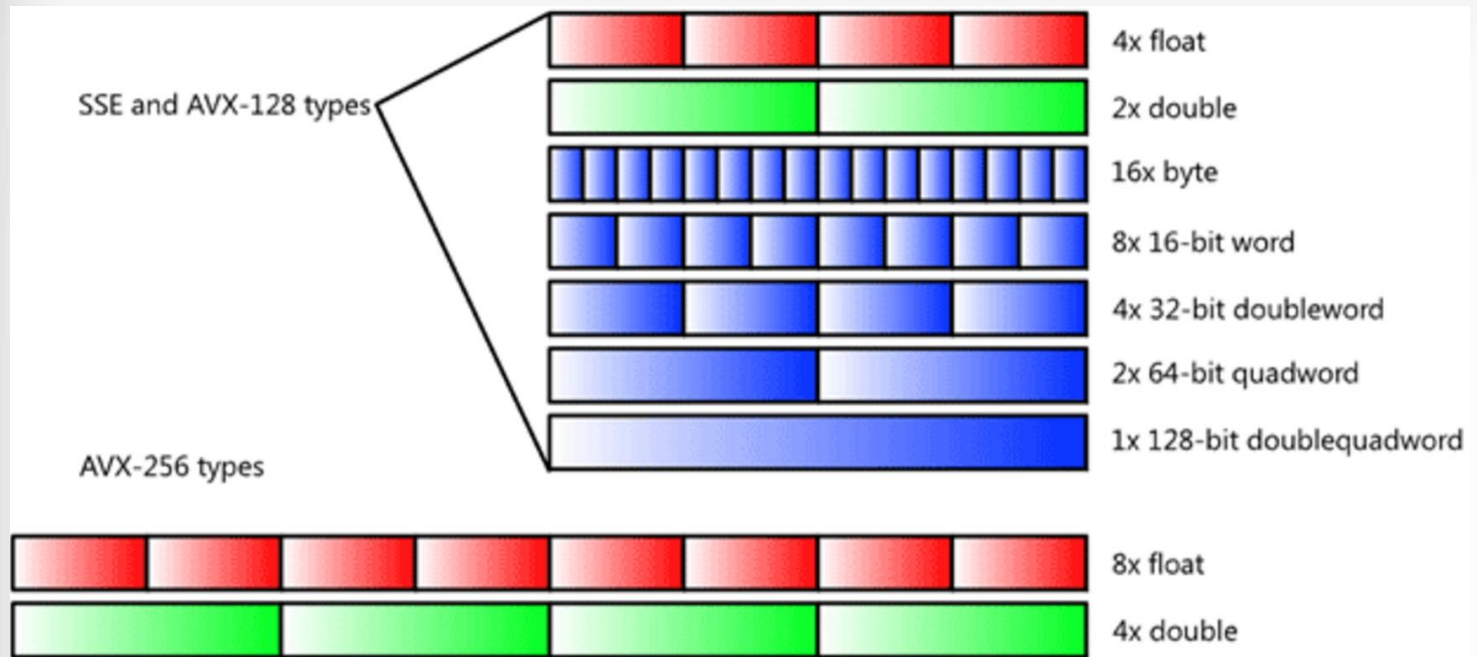
64 bits enteros o *double precision*



# AVX

- ❖ Implementado por Intel y AMD
- ❖ Introduce instrucciones de 256 bits.
- ❖ Mejora la gestión de multi-hilos y multi-nucleos.
- ❖ Los registros XMM se extienden a 256 bits y se llaman YMM.
- ❖ Introduce instrucciones no destructivas.  
(destino distinto de sus fuentes)
- ❖ Las instrucciones pasan de tener dos operandos a 3 o 4 operandos ( $a=a+b$  vs  $d=a+b+c$ ).
- ❖ La optimización la realizan los compiladores en vez de los programadores (que también).

# AVX



**Ilustración de los tipos de datos  
soportados en AVX**

# Evolución AVX

## AVX2

- ❖ Amplia el conjunto de instrucciones
- ❖ Hace que todas las instrucciones SSE se puedan ejecutar en AVX (256 bits)
- ❖ Implementa *Gather* pero no implementa *Scatter*

## AVX512

- ❖ YMM de 256 pasa a ZMM de 512
- ❖ Añade el VL (Vector Length) y sus operaciones para tratar con él.
- ❖ Añade 7 registros de máscara e instrucciones para tratar con ellos.

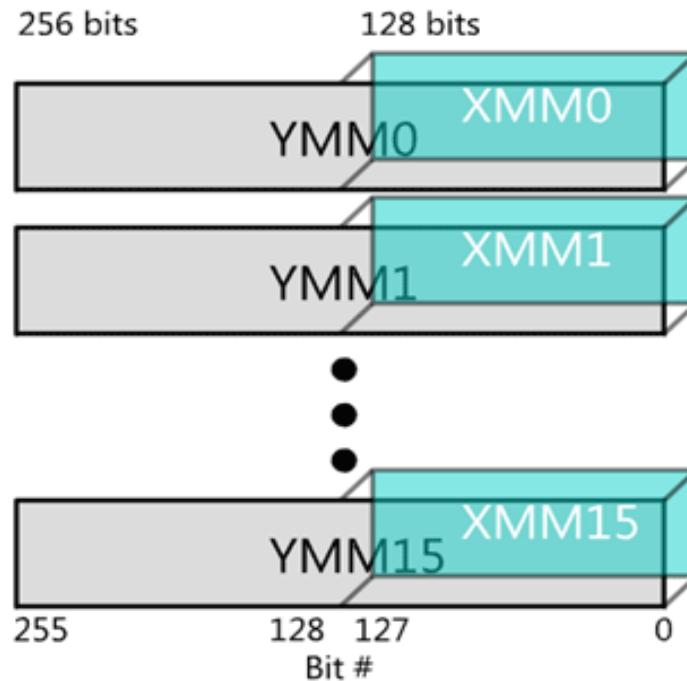
511	255	127	0
ZMM0	YMM0	XMM0	
ZMM1	YMM1	XMM1	
ZMM2	YMM2	XMM2	
ZMM3	YMM3	XMM3	
ZMM4	YMM4	XMM4	
ZMM5	YMM5	XMM5	
ZMM6	YMM6	XMM6	
ZMM7	YMM7	XMM7	
ZMM8	YMM8	XMM8	
ZMM9	YMM9	XMM9	
ZMM10	YMM10	XMM10	
ZMM11	YMM11	XMM11	
ZMM12	YMM12	XMM12	
ZMM13	YMM13	XMM13	
ZMM14	YMM14	XMM14	
ZMM15	YMM15	XMM15	
ZMM16	YMM16	XMM16	
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	

# Registros (review)

- ❖ La evolución extensiones SIMD va acompañada de una evolución de sus registros.
- ❖ La evolución va acompañada de compatibilidad
- ❖ 8 registros MMX de 64 bits que son los mismos que los usados por la FPU.
- ❖ 32 registros ZMM de 512bits
  - ❖ Los 256 bits menos significativos se les identifica como YMM
    - ❖ Los 128 bits menos significativos se les identifica como XMM



# Registros (review)



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	

# Registros (review)

- ❖ Las instrucciones pueden usar los registros como si fuesen elementos escalares, o elementos vectoriales.
- ❖ También pueden usar los registros para tratar números enteros, o en coma flotante, incluso cadenas de caracteres.

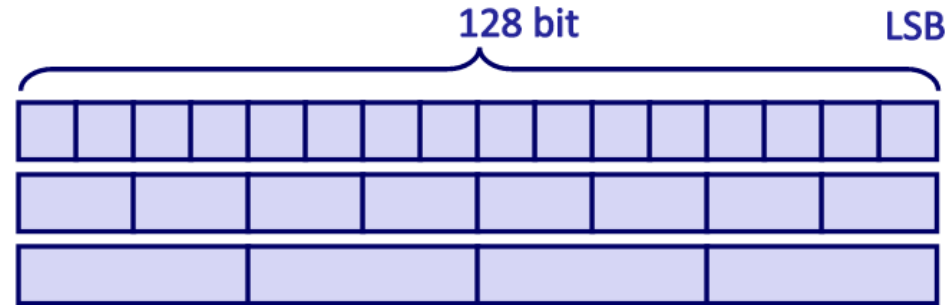
# Registros (review)

## SSE3 Registers

Different data types and associated instructions

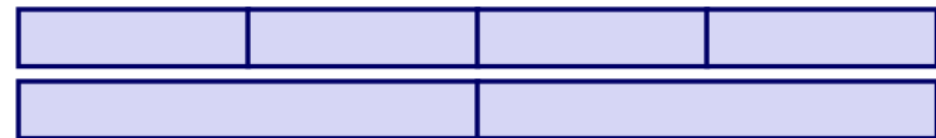
Integer vectors:

- 16-way byte
- 8-way short
- 4-way int



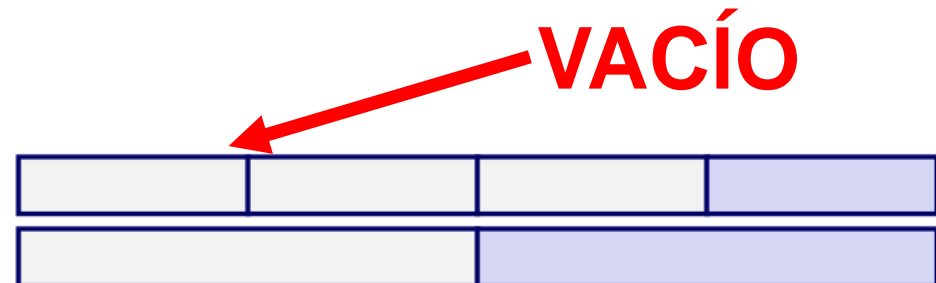
Floating point vectors:

- 4-way single (float)
- 2-way double










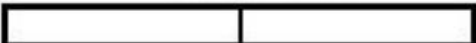
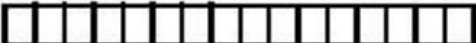

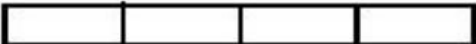




Floating point scalars:

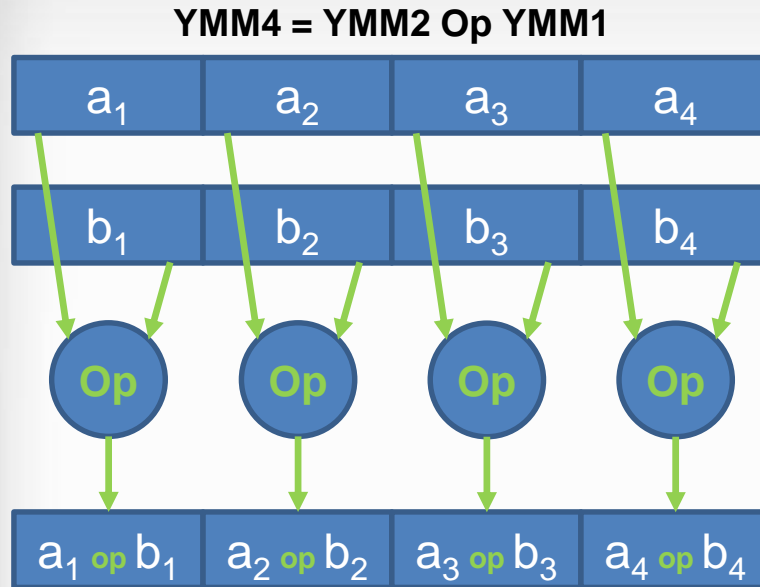
- single
- double



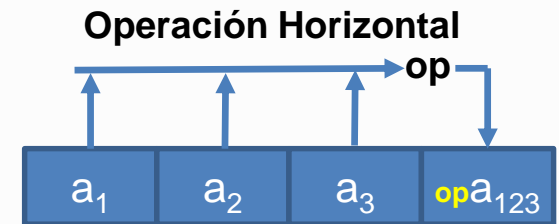
# Registros (review)

SIMD Extension	Register Layout	Data Type
MMX Technology	MMX Registers	
		8 Packed Byte Integers
		4 Packed Word Integers
		2 Packed Doubleword Integers
SSE	MMX Registers	
		8 Packed Byte Integers
		4 Packed Word Integers
		2 Packed Doubleword Integers
SSE2/SSE3	MMX Registers	
		2 Packed Doubleword Integers
		Quadword
	XMM Registers	
		4 Packed Single-Precision Floating-Point Values
	XMM Registers	
		2 Packed Double-Precision Floating-Point Values
		16 Packed Byte Integers
		8 Packed Word Integers
		4 Packed Doubleword Integers
		2 Quadword Integers
		Double Quadword

# Registros (review)



Operación de doble precisión  
con cuatro elementos de un  
vector  
(Por ejemplo suma)



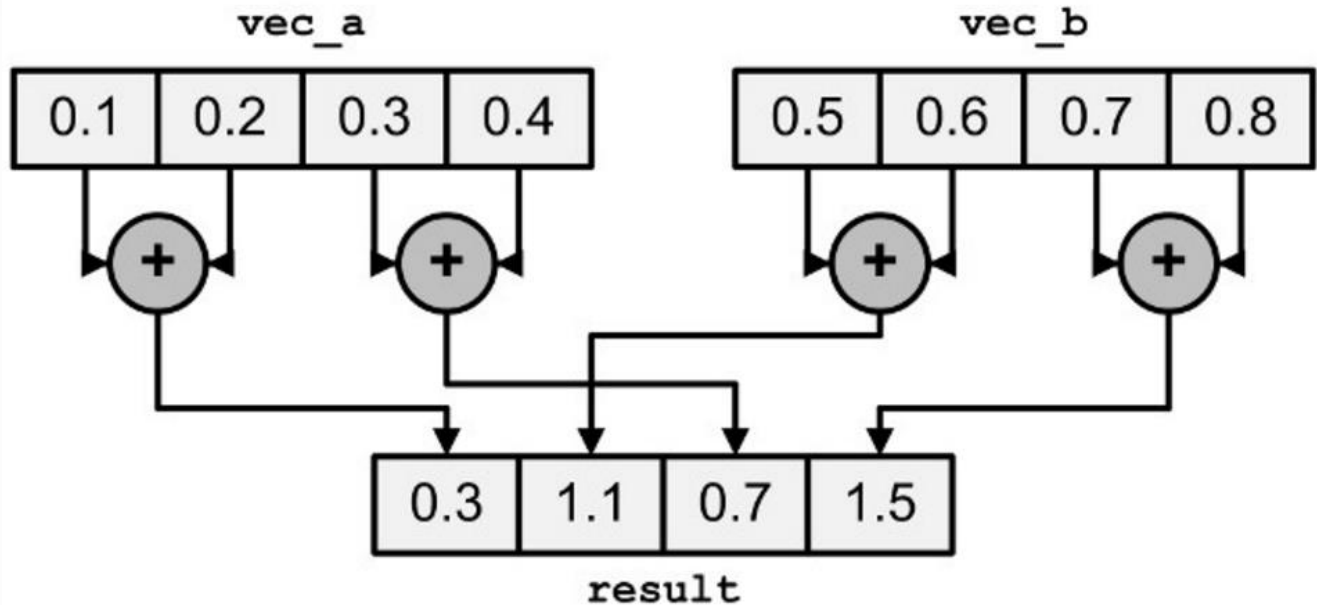
Operación de precisión  
simple con un vector de  
tres elementos  
(Por ejemplo Min)

# Ejemplos de instrucciones

SSE	ADDSS	suma dos valores escalares
	ADDPS	Suma dos vectores
	MINPS	Compara dos vectores
AVX	VADDPD	Suma 4 operandos de 64 bits
	VSUBPD	Resta 4 operandos de 64 bits
	VCMPxx	Compara 4 operandos para xx (EQ, LT, LE,GT....)
Librerías C	_mm256_load_ps	Carga vector (precisión simple) de memoria (alineada)
	_mm256_hadd_epi16	Suma dos vectores de enteros
	_mm256_mullo_epi16	Multiplica enteros y guarda la parte menos significativa
	_mm256_mulhi_epu16	Multiplica enteros y guarda la parte más significativa

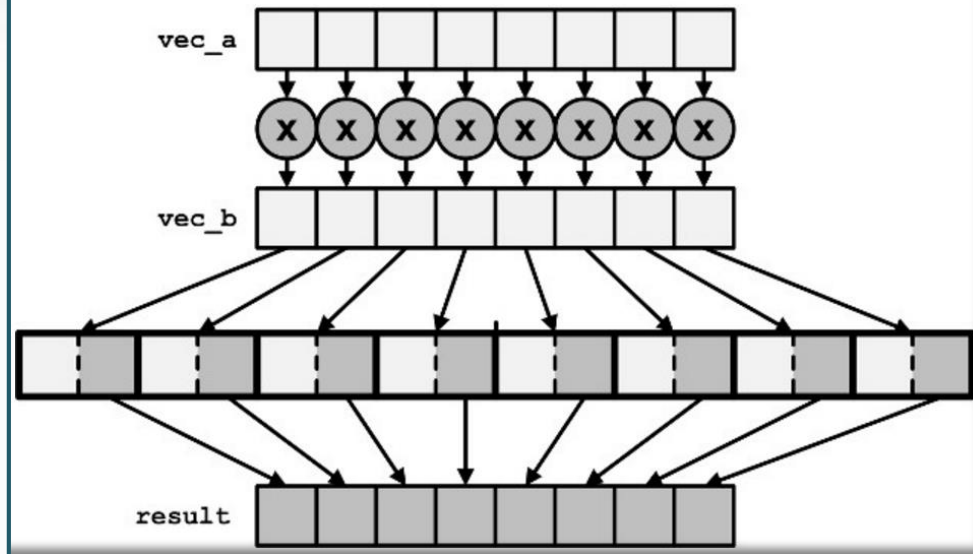
# Ejemplo de instrucciones

```
__m256d result = _mm256_hadd_pd(vec_a, vec_b);
```

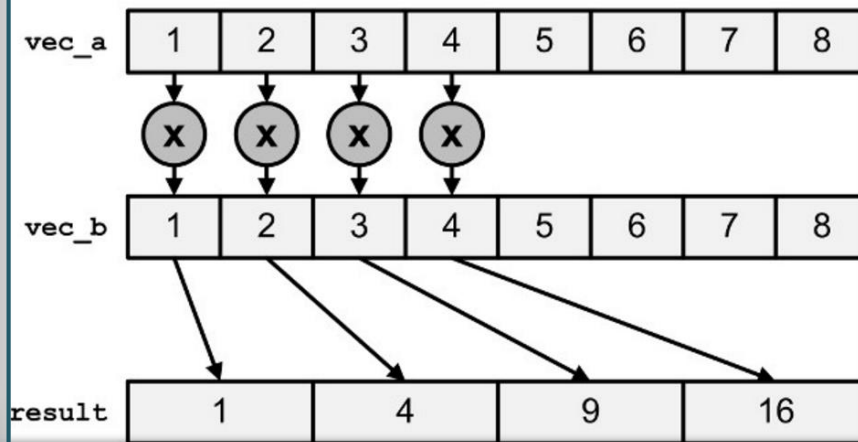


# Ejemplo de instrucciones

```
__m256i result = _mm256_mullo_epi32(vec_a, vec_b);
```



```
__m256i result = _mm256_mul_epi32(vec_a, vec_b);
```





# Ejemplo de uso en C/C++

```
#include <immintrin.h>
#include <stdio.h>

int main() {
    __m256d veca = _mm256_setr_pd(6.0, 6.0, 6.0, 6.0);
    __m256d vecb = _mm256_setr_pd(2.0, 2.0, 2.0, 2.0);
    __m256d vecc = _mm256_setr_pd(7.0, 7.0, 7.0, 7.0);

    /* Alternately subtract and add the third vector
       from the product of the first and second vectors */
    __m256d result = _mm256_fmaddsub_pd (veca, vecb, vecc);

    /* Display the elements of the result vector */
    double* res = (double*)&result;
    printf("%lf %lf %lf %lf\n", res[0], res[1], res[2], res[3]);

    return 0;
}
```

# Ejemplo de uso en ASM

```
//SSE simd function for vectorized multiplication of 2 arrays with single-precision floating point numbers
//1st param pointer on source/destination array, 2nd param 2. source array, 3rd param number of floats per array
void mul_asm(float* out, float* in, unsigned int leng)
{
    unsigned int count, rest;

    //compute if array is big enough for vector operation
    rest = (leng*4)%16;
    count = (leng*4)-rest;

    // vectorized part; 4 floats per loop iteration
    if (count>0){
        __asm __volatile__ (".intel_syntax noprefix\n\t"
            "loop:                                \n\t"
            "movups xmm0,[ebx+ecx] ;loads 4 floats in first register (xmm0)\n\t"
            "movups xmm1,[eax+ecx] ;loads 4 floats in second register (xmm1)\n\t"
            "mulps xmm0,xmm1           ;multiplies both vector registers\n\t"
            "movups [eax+ecx],xmm0      ;write back the result\n\t"
            "sub ecx,16                 ;increase address pointer by 4 floats\n\t"
            "jnz loop                    \n\t"
            ".att_syntax prefix         \n\t"
            : : "a" (out), "b" (in), "c"(count), "d"(rest): "xmm0","xmm1");
    }

    // scalar part; 1 float per loop iteration
    if (rest!=0)
    {
        __asm __volatile__ (".intel_syntax noprefix\n\t"
            "add eax,ecx                \n\t"
            "add ebx,ecx                \n\t"
            "rest:                      \n\t"
            "movss xmm0,[ebx+edx]       ;load 1 float in first register (xmm0)\n\t"
            "movss xmm1,[eax+edx]       ;load 1 float in second register (xmm1)\n\t"
            "mulss xmm0,xmm1            ;multiplies both scalar parts of registers\n\t"
            "movss [eax+edx],xmm0        ;write back the result\n\t"
            "sub edx,4                   \n\t"
            "jnz rest                    \n\t"
            ".att_syntax prefix         \n\t"
            : : "a" (out), "b" (in), "c"(count), "d"(rest): "xmm0","xmm1");
    }
    return;
}
```