SIMD Parte 1

Saturación, Empaquetado, Extensión y Mascaras

David Alejandro González Márquez

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Introducción

- Vamos a resolver algoritmos utililzando **instrucciones vectoriales**.
- Debemos **conocer** las instrucciones que tenemos disponibles.
- y las **técnicas** para pensar algoritmos desde la operatoria vectoriales.

Registros y tipos de datos

- Registros:

  `XMM0` a `XMM15` de 128 bits (16 bytes)

- Tipos de datos:

  Enteros: 8, 16, 32, 64 y 128.

  Float: 32 (Float) y 64 (Double).

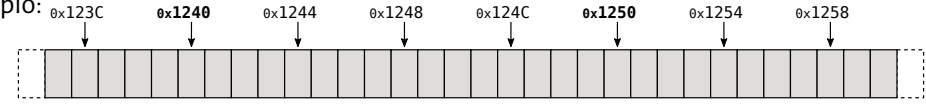
Operaciones Load/Store

MOVD	MOVQ	Move Doubleword/Quadword
MOVSS	MOVSD	Moves a 32bits Single FP/64bits Double FP
MOVDQA	MOVDQU	Moves aligned/unaligned double quadword
MOVAPS	MOVUPS	Moves 4 aligned/unaligned 32bit singles
MOVAPD	MOVUPD	Moves 2 aligned/unaligned 64bit doubles

Operaciones Load/Store

MOVD	MOVQ	Move Doubleword/Quadword
MOVSS	MOVSD	Moves a 32bits Single FP/64bits Double FP
MOVDQA	MOVDQU	Moves aligned/unaligned double quadword
MOVAPS	MOVUPS	Moves 4 aligned/unaligned 32bit singles
MOVAPD	MOVUPD	Moves 2 aligned/unaligned 64bit doubles

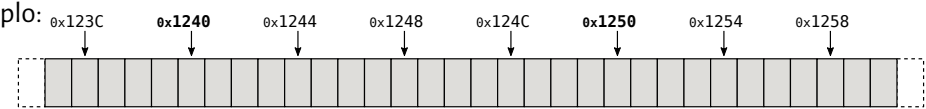
Ejemplo:



Operaciones Load/Store

MOVD	MOVQ	Move Doubleword/Quadword
MOVSS	MOVSD	Moves a 32bits Single FP/64bits Double FP
MOVDQA	MOVDQU	Moves aligned/unaligned double quadword
MOVAPS	MOVUPS	Moves 4 aligned/unaligned 32bit singles
MOVAPD	MOVUPD	Moves 2 aligned/unaligned 64bit doubles

Ejemplo:

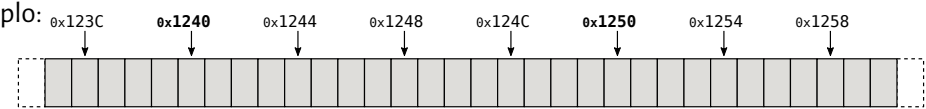


`MOVD [0x123C], xmm0` ✓ `[0x123C]` ← `xmm0(32:0)`

Operaciones Load/Store

MOVD	MOVQ	Move Doubleword/Quadword
MOVSS	MOVSD	Moves a 32bits Single FP/64bits Double FP
MOVDQA	MOVDQU	Moves aligned/unaligned double quadword
MOVAPS	MOVUPS	Moves 4 aligned/unaligned 32bit singles
MOVAPD	MOVUPD	Moves 2 aligned/unaligned 64bit doubles

Ejemplo:

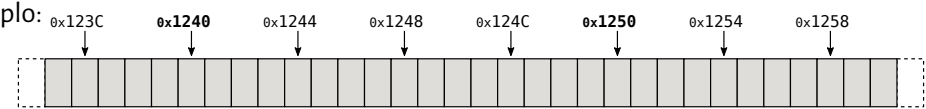



```
MOVD [0x123C], xmm0        ✓   [0x123C] ← xmm0(32:0)
MOVQ xmm0, [0x1245]        ✓   xmm0(64:0) ← [0x1245]
```

Operaciones Load/Store

MOVD	MOVQ	Move Doubleword/Quadword
MOVSS	MOVSD	Moves a 32bits Single FP/64bits Double FP
MOVDQA	MOVDQU	Moves aligned/unaligned double quadword
MOVAPS	MOVUPS	Moves 4 aligned/unaligned 32bit singles
MOVAPD	MOVUPD	Moves 2 aligned/unaligned 64bit doubles

Ejemplo:

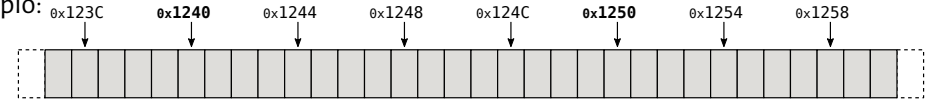


`MOVD [0x123C], xmm0`	✓	`[0x123C]` ← `xmm0(32:0)`
`MOVQ xmm0, [0x1245]`	✓	`xmm0(64:0)` ← `[0x1245]`
`MOVDQA xmm0, [0x1245]`	×	Error dirección no alineada.

Operaciones Load/Store

MOVD	MOVQ	Move Doubleword/Quadword
MOVSS	MOVSD	Moves a 32bits Single FP/64bits Double FP
MOVDQA	MOVDQU	Moves aligned/unaligned double quadword
MOVAPS	MOVUPS	Moves 4 aligned/unaligned 32bit singles
MOVAPD	MOVUPD	Moves 2 aligned/unaligned 64bit doubles

Ejemplo:

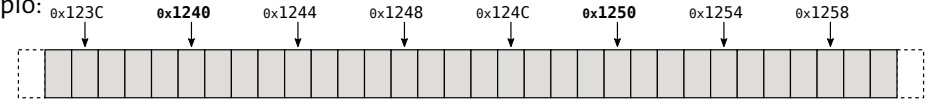



```
MOVD [0x123C], xmm0          ✓  [0x123C] ← xmm0(32:0)
MOVQ xmm0, [0x1245]          ✓  xmm0(64:0) ← [0x1245]
MOVDQA xmm0, [0x1245]        ×  Error dirección no alineada.
MOVDQA [0x1250], xmm0        ✓  [0x1250] ← xmm0(128:0)
```

Operaciones Load/Store

MOVD	MOVQ	Move Doubleword/Quadword
MOVSS	MOVSD	Moves a 32bits Single FP/64bits Double FP
MOVDQA	MOVDQU	Moves aligned/unaligned double quadword
MOVAPS	MOVUPS	Moves 4 aligned/unaligned 32bit singles
MOVAPD	MOVUPD	Moves 2 aligned/unaligned 64bit doubles

Ejemplo:

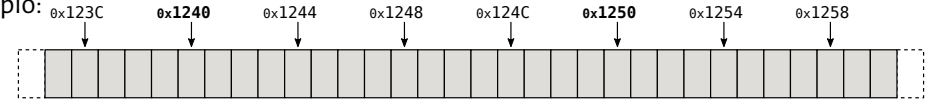


`MOVD [0x123C], xmm0`	✓	`[0x123C]` ← `xmm0(32:0)`
`MOVQ xmm0, [0x1245]`	✓	`xmm0(64:0)` ← `[0x1245]`
`MOVDQA xmm0, [0x1245]`	×	Error dirección no alineada.
`MOVDQA [0x1250], xmm0`	✓	`[0x1250]` ← `xmm0(128:0)`
`MOVSS xmm0, [0x1248]`	✓	`xmm0(32:0)` ← `[0x1248]` ; sobre punto flotante

Operaciones Load/Store

MOVD	MOVQ	Move Doubleword/Quadword
MOVSS	MOVSD	Moves a 32bits Single FP/64bits Double FP
MOVDQA	MOVDQU	Moves aligned/unaligned double quadword
MOVAPS	MOVUPS	Moves 4 aligned/unaligned 32bit singles
MOVAPD	MOVUPD	Moves 2 aligned/unaligned 64bit doubles

Ejemplo:



`MOVD [0x123C], xmm0`	✓	`[0x123C]` ← `xmm0(32:0)`
`MOVQ xmm0, [0x1245]`	✓	`xmm0(64:0)` ← `[0x1245]`
`MOVDQA xmm0, [0x1245]`	×	Error dirección no alineada.
`MOVDQA [0x1250], xmm0`	✓	`[0x1250]` ← `xmm0(128:0)`
`MOVSS xmm0, [0x1248]`	✓	`xmm0(32:0)` ← `[0x1248]` ; sobre punto flotante
`MOVUPS [0x1258], xmm0`	✓	`[0x1258]` ← `xmm0(128:0)` ; sobre punto flotante

Operaciones Load/Store

PMOVSXBW	PMOVZXBW	packed sign/zero extension byte to word
PMOVSXBD	PMOVZXBD	packed sign/zero extension byte to dword
PMOVSXBQ	PMOVZXBQ	packed sign/zero extension byte to qword
PMOVSXWD	PMOVZXWD	packed sign/zero extension word to dword
PMOVSXWQ	PMOVZXWQ	packed sign/zero extension word to qword
PMOVSXDQ	PMOVZXDQ	packed sign/zero extension word to dqword

Operaciones Load/Store

PMOVSXBW	PMOVZXBW	packed sign/zero extension byte to word
PMOVSXBD	PMOVZXBD	packed sign/zero extension byte to dword
PMOVSXBQ	PMOVZXBQ	packed sign/zero extension byte to qword
PMOVSXWD	PMOVZXWD	packed sign/zero extension word to dword
PMOVSXWQ	PMOVZXWQ	packed sign/zero extension word to qword
PMOVSXDQ	PMOVZXDQ	packed sign/zero extension word to dqword

Operaciones Load/Store

PMOVSXBW	PMOVZXBW	packed sign/zero extension byte to word
PMOVSXBD	PMOVZXBD	packed sign/zero extension byte to dword
PMOVSXBQ	PMOVZXBQ	packed sign/zero extension byte to qword
PMOVSXWD	PMOVZXWD	packed sign/zero extension word to dword
PMOVSXWQ	PMOVZXWQ	packed sign/zero extension word to qword
PMOVSXDQ	PMOVZXDQ	packed sign/zero extension word to dqword

Ejemplos:

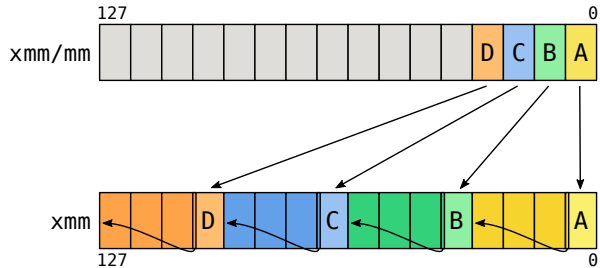
Operaciones Load/Store

PMOVSXBW	PMOVZXBW	packed sign/zero extension byte to word
PMOVSXBD	PMOVZXBD	packed sign/zero extension byte to dword
PMOVSXBQ	PMOVZXBQ	packed sign/zero extension byte to qword
PMOVSXWD	PMOVZXWD	packed sign/zero extension word to dword
PMOVSXWQ	PMOVZXWQ	packed sign/zero extension word to qword
PMOVSXDQ	PMOVZXDQ	packed sign/zero extension word to dqword

Ejemplos:

```
PMOVSXBD xmm0, xmm0
```

✓

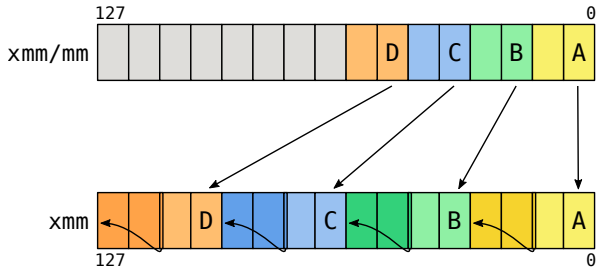


Operaciones Load/Store

PMOVSXBW	PMOVZXBW	packed sign/zero extension byte to word
PMOVSXBD	PMOVZXBD	packed sign/zero extension byte to dword
PMOVSXBQ	PMOVZXBQ	packed sign/zero extension byte to qword
PMOVSXWD	PMOVZXWD	packed sign/zero extension word to dword
PMOVSXWQ	PMOVZXWQ	packed sign/zero extension word to qword
PMOVSXDQ	PMOVZXDQ	packed sign/zero extension word to dqword

Ejemplos:

```
PMOVSXBD xmm0, xmm0      ✓
PMOVZXWD xmm0, [data]    ✓
```

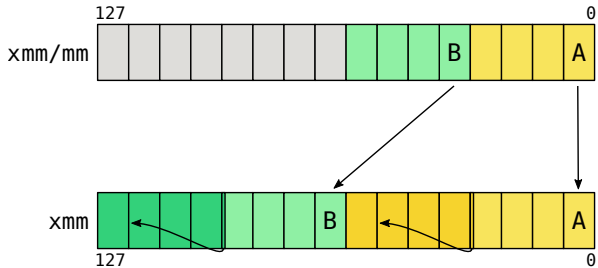


Operaciones Load/Store

PMOVSXBW	PMOVZXBW	packed sign/zero extension byte to word
PMOVSXBD	PMOVZXBD	packed sign/zero extension byte to dword
PMOVSXBQ	PMOVZXBQ	packed sign/zero extension byte to qword
PMOVSXWD	PMOVZXWD	packed sign/zero extension word to dword
PMOVSXWQ	PMOVZXWQ	packed sign/zero extension word to qword
PMOVSXDQ	PMOVZXDQ	packed sign/zero extension word to dqword

Ejemplos:

```
PMOVSXBD xmm0, xmm0      ✓
PMOVZXWD xmm0, [data]    ✓
PMOVZXDQ xmm0, xmm1      ✓
```



Operaciones Load/Store

PMOVSXBW	PMOVZXBW	packed sign/zero extension byte to word
PMOVSXBD	PMOVZXBD	packed sign/zero extension byte to dword
PMOVSXBQ	PMOVZXBQ	packed sign/zero extension byte to qword
PMOVSXWD	PMOVZXWD	packed sign/zero extension word to dword
PMOVSXWQ	PMOVZXWQ	packed sign/zero extension word to qword
PMOVSXDQ	PMOVZXDQ	packed sign/zero extension word to dqword

Ejemplos:

```
PMOVSXBD xmm0, xmm0      ✓
PMOVZXWD xmm0, [data]    ✓
PMOVZXDQ xmm0, xmm1      ✓
PMOVZXQD xmm0, xmm0      ×   Instrucción invalida.
```

Operaciones Load/Store

PMOVSXBW	PMOVZXBW	packed sign/zero extension byte to word
PMOVSXBD	PMOVZXBD	packed sign/zero extension byte to dword
PMOVSXBQ	PMOVZXBQ	packed sign/zero extension byte to qword
PMOVSXWD	PMOVZXWD	packed sign/zero extension word to dword
PMOVSXWQ	PMOVZXWQ	packed sign/zero extension word to qword
PMOVSXDQ	PMOVZXDQ	packed sign/zero extension word to dqword

Ejemplos:

```
PMOVSXBD xmm0, xmm0      ✓
PMOVZXWD xmm0, [data]    ✓
PMOVZXDQ xmm0, xmm1      ✓
PMOVZXQD xmm0, xmm0      ×   Instrucción invalida.
PMOVSXBD [data], xmm0    ×   Modo de direccionamiento invalido.
```

Operaciones Aritméticas

PADDB	PADDW	PADDD	PADDQ	Add Integer
PSUBB	PSUBW	PSUBD	PSUBQ	Sub Integer
PMULHW	PMULLW			Mul Integer Word
PMULHD	PMULLD			Mul Integer Dword
PMINSB	PMAXSB	PMINUB	PMAXUB	Max and Min Integer
PMINSW	PMAXSW	PMINUW	PMAXUW	Max and Min Integer
PMINSD	PMAXSD	PMINUD	PMAXUD	Max and Min Integer

Operaciones Aritméticas

PADDB	PADDW	PADDD	PADDQ	Add Integer
PSUBB	PSUBW	PSUBD	PSUBQ	Sub Integer
PMULHW	PMULLW			Mul Integer Word
PMULHD	PMULLD			Mul Integer Dword
PMINSB	PMAXSB	PMINUB	PMAXUB	Max and Min Integer
PMINSW	PMAXSW	PMINUW	PMAXUW	Max and Min Integer
PMINSD	PMAXSD	PMINUD	PMAXUD	Max and Min Integer

Ejemplos:

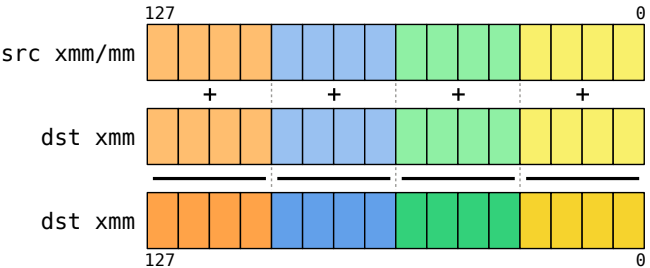
Operaciones Aritméticas

PADDB	PADDW	PADDD	PADDQ	Add Integer
PSUBB	PSUBW	PSUBD	PSUBQ	Sub Integer
PMULHW	PMULLW			Mul Integer Word
PMULHD	PMULLD			Mul Integer Dword
PMINSB	PMAXSB	PMINUB	PMAXUB	Max and Min Integer
PMINSW	PMAXSW	PMINUW	PMAXUW	Max and Min Integer
PMINSD	PMAXSD	PMINUD	PMAXUD	Max and Min Integer

Ejemplos:

```
PADDD xmm0, xmm1
```

✓

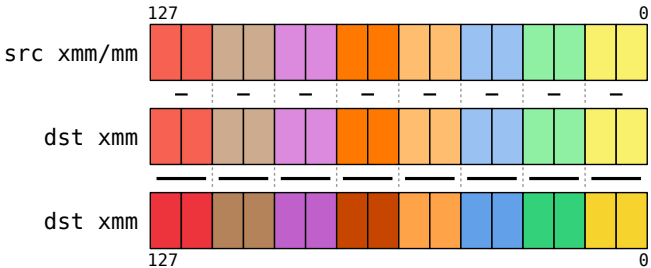


Operaciones Aritméticas

PADDB	PADDW	PADDD	PADDQ	Add Integer
PSUBB	PSUBW	PSUBD	PSUBQ	Sub Integer
PMULHW	PMULLW			Mul Integer Word
PMULHD	PMULLD			Mul Integer Dword
PMINSB	PMAXSB	PMINUB	PMAXUB	Max and Min Integer
PMINSW	PMAXSW	PMINUW	PMAXUW	Max and Min Integer
PMINSD	PMAXSD	PMINUD	PMAXUD	Max and Min Integer

Ejemplos:

```
PADDD xmm0, xmm1      ✓
PSUBW xmm0, [data]    ✓
```

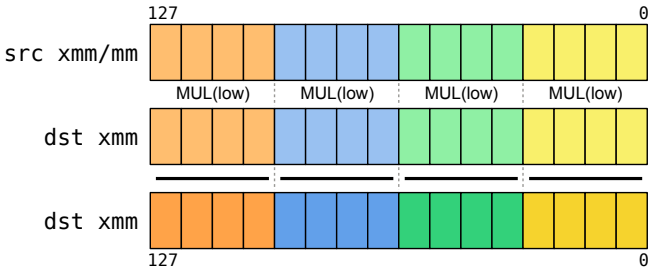


Operaciones Aritméticas

PADDB	PADDW	PADDD	PADDQ	Add Integer
PSUBB	PSUBW	PSUBD	PSUBQ	Sub Integer
PMULHW	PMULLW			Mul Integer Word
PMULHD	PMULLD			Mul Integer Dword
PMINSB	PMAXSB	PMINUB	PMAXUB	Max and Min Integer
PMINSW	PMAXSW	PMINUW	PMAXUW	Max and Min Integer
PMINSD	PMAXSD	PMINUD	PMAXUD	Max and Min Integer

Ejemplos:

```
PADDD xmm0, xmm1        ✓
PSUBW xmm0, [data]      ✓
PMULLD xmm0, xmm1       ✓
```

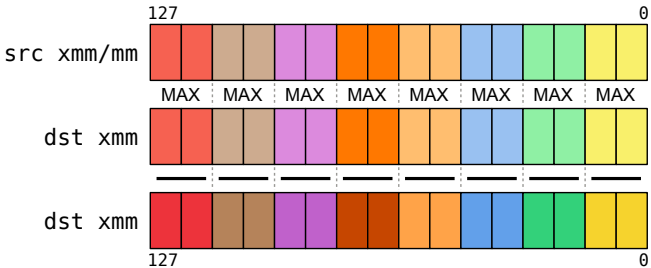


Operaciones Aritméticas

PADDB	PADDW	PADDD	PADDQ	Add Integer
PSUBB	PSUBW	PSUBD	PSUBQ	Sub Integer
PMULHW	PMULLW			Mul Integer Word
PMULHD	PMULLD			Mul Integer Dword
PMINSB	PMAXSB	PMINUB	PMAXUB	Max and Min Integer
PMINSW	PMAXSW	PMINUW	PMAXUW	Max and Min Integer
PMINSD	PMAXSD	PMINUD	PMAXUD	Max and Min Integer

Ejemplos:

```
PADDD xmm0, xmm1          ✓
PSUBW xmm0, [data]        ✓
PMULLD xmm0, xmm1         ✓
PMAXSW xmm0, [data]       ✓
```



Operaciones Aritméticas

PADDB	PADDW	PADDD	PADDQ	Add Integer
PSUBB	PSUBW	PSUBD	PSUBQ	Sub Integer
PMULHW	PMULLW			Mul Integer Word
PMULHD	PMULLD			Mul Integer Dword
PMINSB	PMAXSB	PMINUB	PMAXUB	Max and Min Integer
PMINSW	PMAXSW	PMINUW	PMAXUW	Max and Min Integer
PMINSD	PMAXSD	PMINUD	PMAXUD	Max and Min Integer

Ejemplos:

```
PADDD xmm0, xmm1        ✓
PSUBW xmm0, [data]      ✓
PMULLD xmm0, xmm1       ✓
PMAXSW xmm0, [data]     ✓
PMINSB [data], xmm0     ×   Modo de direccionamiento invalido.
```

Operaciones Aritméticas

PABSB	Absolute for 8 bit Integers
PABSW	Absolute for 16 bit Integers
PABSD	Absolute for 32 bit Integers

Operaciones Aritméticas

PABSB	Absolute for 8 bit Integers
PABSW	Absolute for 16 bit Integers
PABSD	Absolute for 32 bit Integers

Ejemplos:

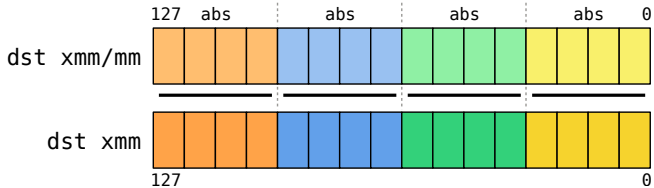
Operaciones Aritméticas

PABSB	Absolute for 8 bit Integers
PABSW	Absolute for 16 bit Integers
PABSD	Absolute for 32 bit Integers

Ejemplos:

```
PABSD xmm0, xmm0
```

✓

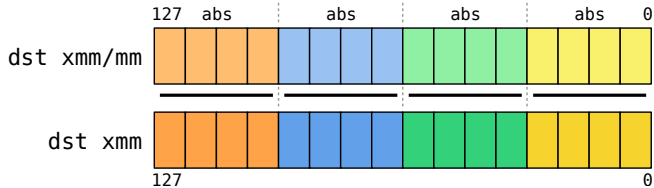


Operaciones Aritméticas

PABSB	Absolute for 8 bit Integers
PABSW	Absolute for 16 bit Integers
PABSD	Absolute for 32 bit Integers

Ejemplos:

```
PABSD xmm0, xmm0      ✓
PABSD xmm0, [data]    ✓
```



Operaciones Aritméticas

PABSB	Absolute for 8 bit Integers
PABSW	Absolute for 16 bit Integers
PABSD	Absolute for 32 bit Integers

Ejemplos:

```
PABSD xmm0, xmm0      ✓
PABSD xmm0, [data]    ✓
PABSD [data], xmm0    ×   Modo de direccionamiento invalido.
```

Ejemplo

Suma Uno

Dado un vector de n enteros sin signo de 16 bits. Incrementa en 1 unidad cada uno y almacena el resultado en un vector de 16 bits.
Considerar $n \equiv 0 \ (mod\ 8)$.

```
void suma1(uint16_t *vector, uint16_t *resultado, uint8_t n)
```

Suma Uno

```
section .rodata
uno: times 8 dw 1

section .text

suma1: ; rdi = vector, rsi = resultado, dx = n
```

Suma Uno

```
section .rodata
uno: times 8 dw 1

section .text

suma1: ; rdi = vector, rsi = resultado, dx = n
    push rbp
    mov rbp,rsp
```

Suma Uno

```
section .rodata
uno: times 8 dw 1

section .text

suma1: ; rdi = vector, rsi = resultado, dx = n
    push rbp
    mov rbp,rsp
    movzx rcx, dx
    shr ecx, 3          ; divido por 8
```

Suma Uno

```
section .rodata
uno: times 8 dw 1

section .text

suma1: ; rdi = vector, rsi = resultado, dx = n
    push rbp
    mov rbp,rsp
    movzx rcx, dx
    shr ecx, 3          ; divido por 8
    movdqu xmm8, [uno] ; xmm8 = | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
```

Suma Uno

```
section .rodata
uno: times 8 dw 1

section .text

suma1: ; rdi = vector, rsi = resultado, dx = n
    push rbp
    mov rbp,rsp
    movzx rcx, dx
    shr ecx, 3          ; divido por 8
    movdqu xmm8, [uno] ; xmm8 = | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
    .ciclo:
```

Suma Uno

```
section .rodata
uno: times 8 dw 1

section .text

suma1: ; rdi = vector, rsi = resultado, dx = n
    push rbp
    mov rbp,rsp
    movzx rcx, dx
    shr ecx, 3          ; divido por 8
    movdqu xmm8, [uno] ; xmm8 = | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
    .ciclo:
        movdqu xmm0, [rdi] ; xmm0 = | d7 | d6 | d5 | d4 | d3 | d2 | d1 | d0 |
```

Suma Uno

```
section .rodata
uno: times 8 dw 1

section .text

suma1: ; rdi = vector, rsi = resultado, dx = n
    push rbp
    mov rbp,rsp
    movzx rcx, dx
    shr ecx, 3          ; divido por 8
    movdqu xmm8, [uno] ; xmm8 = | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
    .ciclo:
        movdqu xmm0, [rdi] ; xmm0 = | d7 | d6 | d5 | d4 | d3 | d2 | d1 | d0 |
        paddw  xmm0, xmm8  ; xmm0 = |d7+1|d6+1|d5+1|d4+1|d3+1|d2+1|d1+1|d0+1|
```

Suma Uno

```
section .rodata
uno: times 8 dw 1

section .text

suma1: ; rdi = vector, rsi = resultado, dx = n
    push rbp
    mov rbp,rsp
    movzx rcx, dx
    shr ecx, 3          ; divido por 8
    movdqu xmm8, [uno] ; xmm8 = | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
    .ciclo:
        movdqu xmm0, [rdi] ; xmm0 = | d7 | d6 | d5 | d4 | d3 | d2 | d1 | d0 |
        paddw  xmm0, xmm8  ; xmm0 = |d7+1|d6+1|d5+1|d4+1|d3+1|d2+1|d1+1|d0+1|
        movdqu [rsi], xmm0
```

Suma Uno

```
section .rodata
uno: times 8 dw 1

section .text

suma1: ; rdi = vector, rsi = resultado, dx = n
    push rbp
    mov rbp,rsp
    movzx rcx, dx
    shr ecx, 3          ; divido por 8
    movdqu xmm8, [uno] ; xmm8 = | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
    .ciclo:
        movdqu xmm0, [rdi] ; xmm0 = | d7 | d6 | d5 | d4 | d3 | d2 | d1 | d0 |
        paddw  xmm0, xmm8  ; xmm0 = |d7+1|d6+1|d5+1|d4+1|d3+1|d2+1|d1+1|d0+1|
        movdqu [rsi], xmm0
        add rdi, 16
        add rsi, 16
```

Suma Uno

```
section .rodata
uno: times 8 dw 1

section .text

suma1: ; rdi = vector, rsi = resultado, dx = n
    push rbp
    mov rbp,rsp
    movzx rcx, dx
    shr ecx, 3          ; divido por 8
    movdqu xmm8, [uno] ; xmm8 = | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
    .ciclo:
        movdqu xmm0, [rdi] ; xmm0 = | d7 | d6 | d5 | d4 | d3 | d2 | d1 | d0 |
        paddw  xmm0, xmm8  ; xmm0 = |d7+1|d6+1|d5+1|d4+1|d3+1|d2+1|d1+1|d0+1|
        movdqu [rsi], xmm0
        add rdi, 16
        add rsi, 16
    loop .ciclo
    pop rbp
    ret
```

Ejemplo

Suma Dos

Dado un vector de n enteros con signo de 16 bits. Incrementa en 2 unidades cada uno y almacena el resultado en un vector de 32 bits.
Considerar $n \equiv 0 \pmod 8$.

```
void suma2(int16_t *vector, int32_t *resultado, uint8_t n)
```

Suma Dos

```
section .rodata
dos: times 4 dd 2

section .text

suma2: ; rdi = vector, rsi = resultado, dx = n
    push rbp
    mov rbp,rsp
```

Suma Dos

```
section .rodata
dos: times 4 dd 2

section .text

suma2: ; rdi = vector, rsi = resultado, dx = n
    push rbp
    mov rbp,rsp
    movzx rcx, dx
    shr ecx, 2          ; divido por 4
```

Suma Dos

```
section .rodata
dos: times 4 dd 2

section .text

suma2: ; rdi = vector, rsi = resultado, dx = n
    push rbp
    mov rbp,rsp
    movzx rcx, dx
    shr ecx, 2          ; divido por 4
    movdqu xmm8, [dos] ; xmm8 = | 2 | 2 | 2 | 2 |
```

Suma Dos

```
section .rodata
dos: times 4 dd 2

section .text

suma2: ; rdi = vector, rsi = resultado, dx = n
    push rbp
    mov rbp,rsp
    movzx rcx, dx
    shr ecx, 2          ; divido por 4
    movdqu xmm8, [dos] ; xmm8 = | 2 | 2 | 2 | 2 |
    .ciclo:
```

Suma Dos

```
section .rodata
dos: times 4 dd 2

section .text

suma2: ; rdi = vector, rsi = resultado, dx = n
    push rbp
    mov rbp,rsp
    movzx rcx, dx
    shr ecx, 2          ; divido por 4
    movdqu xmm8, [dos] ; xmm8 = | 2 | 2 | 2 | 2 |
    .ciclo:
        pmovsxwd xmm0, [rdi]  ; xmm0 = | d3 | d2 | d1 | d0 |
```

Suma Dos

```
section .rodata
dos: times 4 dd 2

section .text

suma2: ; rdi = vector, rsi = resultado, dx = n
    push rbp
    mov rbp,rsp
    movzx rcx, dx
    shr ecx, 2          ; divido por 4
    movdqu xmm8, [dos] ; xmm8 = | 2 | 2 | 2 | 2 |
    .ciclo:
        pmovsxwd xmm0, [rdi]  ; xmm0 = | d3 | d2 | d1 | d0 |
        paddd  xmm0, xmm8     ; xmm0 = |d3+2|d2+2|d1+2|d0+2|
        movdqu [rsi], xmm0
        add rdi, 8
        add rsi, 16
    loop .ciclo
```

Suma Dos

```
section .rodata
dos: times 4 dd 2

section .text

suma2: ; rdi = vector, rsi = resultado, dx = n
    push rbp
    mov rbp,rsp
    movzx rcx, dx
    shr ecx, 2          ; divido por 4
    movdqu xmm8, [dos] ; xmm8 = | 2 | 2 | 2 | 2 |
    .ciclo:
        pmovsxwd xmm0, [rdi]  ; xmm0 = | d3 | d2 | d1 | d0 |
        paddd  xmm0, xmm8     ; xmm0 = |d3+2|d2+2|d1+2|d0+2|
        movdqu [rsi], xmm0
        add rdi, 8
        add rsi, 16
    loop .ciclo
    pop rbp
    ret
```

Operaciones Aritméticas

PADDSB	PADDSW	Add Int saturation
PADDUSB	PADDUSW	Add Int unsigned saturation
PSUBSB	PSUBSW	Sub Int saturation
PSUBUSB	PSUBUSW	Sub Int unsigned saturation

Operaciones Aritméticas

PADDSB	PADDSW	Add Int saturation
PADDUSB	PADDUSW	Add Int unsigned saturation
PSUBSB	PSUBSW	Sub Int saturation
PSUBUSB	PSUBUSW	Sub Int unsigned saturation

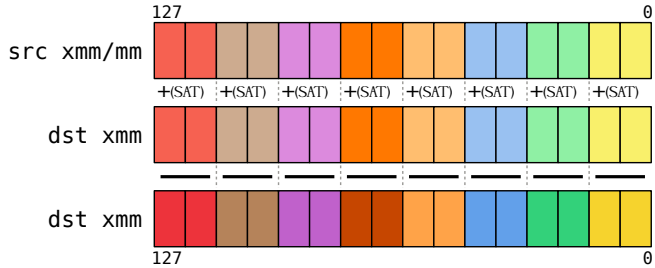
Ejemplos:

Operaciones Aritméticas

PADDSB	PADDSW	Add Int saturation
PADDUSB	PADDUSW	Add Int unsigned saturation
PSUBSB	PSUBSW	Sub Int saturation
PSUBUSB	PSUBUSW	Sub Int unsigned saturation

Ejemplos:

`PADDSW xmm0, xmm0` ✓

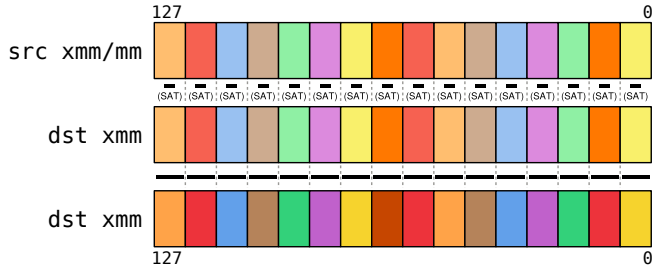


Operaciones Aritméticas

PADDSB	PADDSW	Add Int saturation
PADDUSB	PADDUSW	Add Int unsigned saturation
PSUBSB	PSUBSW	Sub Int saturation
PSUBUSB	PSUBUSW	Sub Int unsigned saturation

Ejemplos:

```
PADDSW xmm0, xmm0          ✓
PSUBUSB xmm0, [data]       ✓
```



Operaciones Aritméticas

PADDSB	PADDSW	Add Int saturation
PADDUSB	PADDUSW	Add Int unsigned saturation
PSUBSB	PSUBSW	Sub Int saturation
PSUBUSB	PSUBUSW	Sub Int unsigned saturation

Ejemplos:

```
PADDSW xmm0, xmm0        ✓
PSUBUSB xmm0, [data]     ✓
PSUBSW [data], xmm0      ×   Modo de direccionamiento invalido.
```

Ejemplo

Suma Tres

Dado un vector de n enteros con signo de 16 bits. Incrementa en 3 unidades cada uno y almacena el resultado en el mismo vector de forma saturada.
Considerar $n \equiv 0 \ (mod\ 8)$.

```
void suma3(int16_t *vector, uint8_t n)
```

Suma Tres

```
section .rodata
tres: times 8 dw 3

section .text

suma3: ; rdi = vector, rsi = n
    push rbp
    mov rbp,rsp
    movzx rcx, si
    shr ecx, 3              ; divido por 8
    movdqu xmm8, [tres]     ; xmm8 = | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
```

Suma Tres

```
section .rodata
tres: times 8 dw 3

section .text

suma3: ; rdi = vector, rsi = n
    push rbp
    mov rbp,rsp
    movzx rcx, si
    shr ecx, 3              ; divido por 8
    movdqu xmm8, [tres]     ; xmm8 = | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

    .ciclo:
        movdqu xmm0, [rdi]  ; xmm0 = | d7 | d6 | d5 | d4 | d3 | d2 | d1 | d0 |
```

Suma Tres

```
section .rodata
tres: times 8 dw 3

section .text

suma3: ; rdi = vector, rsi = n
    push rbp
    mov rbp,rsp
    movzx rcx, si
    shr ecx, 3              ; divido por 8
    movdqu xmm8, [tres]     ; xmm8 = | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
    .ciclo:
        movdqu xmm0, [rdi]  ; xmm0 = | d7 | d6 | d5 | d4 | d3 | d2 | d1 | d0 |
        paddsw xmm0, xmm8   ; xmm0 = |d7+3|d6+3|d5+3|d4+3|d3+3|d2+3|d1+3|d0+3|
```

Suma Tres

```
section .rodata
tres: times 8 dw 3

section .text

suma3: ; rdi = vector, rsi = n
    push rbp
    mov rbp,rsp
    movzx rcx, si
    shr ecx, 3              ; divido por 8
    movdqu xmm8, [tres]     ; xmm8 = | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

    .ciclo:
        movdqu xmm0, [rdi]  ; xmm0 = | d7 | d6 | d5 | d4 | d3 | d2 | d1 | d0 |

        paddsw xmm0, xmm8   ; xmm0 = |d7+3|d6+3|d5+3|d4+3|d3+3|d2+3|d1+3|d0+3|

        movdqu [rdi], xmm0
        add rdi, 16
    loop .ciclo
```

Suma Tres

```
section .rodata
tres: times 8 dw 3

section .text

suma3: ; rdi = vector, rsi = n
    push rbp
    mov rbp,rsp
    movzx rcx, si
    shr ecx, 3              ; divido por 8
    movdqu xmm8, [tres]     ; xmm8 = | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

    .ciclo:
        movdqu xmm0, [rdi]  ; xmm0 = | d7 | d6 | d5 | d4 | d3 | d2 | d1 | d0 |

        paddsw xmm0, xmm8   ; xmm0 = |d7+3|d6+3|d5+3|d4+3|d3+3|d2+3|d1+3|d0+3|

        movdqu [rdi], xmm0
        add rdi, 16
    loop .ciclo

    pop rbp
    ret
```

Ejemplo

Incrementar Brillo

Dado una imagen 32x32 pixeles de un byte en escala de grises. Incrementar el brillo de la misma en 10 unidades.

```
void incrementarBrillo10(uint8_t *imagen)
```

Incrementar Brillo

```
section .rodata
diez: times 16 db 10

section .text

incrementarBrillo10: ; rdi = imagen
    push rbp
    mov rbp,rsp
```

Incrementar Brillo

```
section .rodata
diez: times 16 db 10

section .text

incrementarBrillo10: ; rdi = imagen
    push rbp
    mov rbp,rsp

    mov rcx, (32*32 >> 4)
```

Incrementar Brillo

```
section .rodata
diez: times 16 db 10

section .text

incrementarBrillo10: ; rdi = imagen
    push rbp
    mov rbp,rsp

    mov rcx, (32*32 >> 4)

    movdqu xmm8, [diez]     ; xmm0 = |  10 | ... |  10 |
```

Incrementar Brillo

```
section .rodata
diez: times 16 db 10

section .text

incrementarBrillo10: ; rdi = imagen
    push rbp
    mov rbp,rsp

    mov rcx, (32*32 >> 4)

    movdqu xmm8, [diez]     ; xmm0 = |  10 | ... |  10 |

    .ciclo:
        movdqu xmm0, [rdi]  ; xmm0 = |  d15 | ... |  d0 |
```

Incrementar Brillo

```
section .rodata
diez: times 16 db 10

section .text

incrementarBrillo10: ; rdi = imagen
    push rbp
    mov rbp,rsp

    mov rcx, (32*32 >> 4)

    movdqu xmm8, [diez]     ; xmm0 = |  10 | ... |  10 |

    .ciclo:
        movdqu xmm0, [rdi]  ; xmm0 = |  d15 | ... |  d0 |

        paddusb xmm0, xmm8  ; xmm0 = |d15+10| ... |d0+10|
```

Incrementar Brillo

```
section .rodata
diez: times 16 db 10

section .text

incrementarBrillo10: ; rdi = imagen
    push rbp
    mov rbp,rsp

    mov rcx, (32*32 >> 4)

    movdqu xmm8, [diez]     ; xmm0 = |  10 | ... |  10 |

    .ciclo:
        movdqu xmm0, [rdi]  ; xmm0 = |  d15 | ... |  d0 |

        paddusb xmm0, xmm8  ; xmm0 = |d15+10| ... |d0+10|

        movdqu [rdi], xmm0
```

Incrementar Brillo

```
section .rodata
diez: times 16 db 10

section .text

incrementarBrillo10: ; rdi = imagen
    push rbp
    mov rbp,rsp

    mov rcx, (32*32 >> 4)

    movdqu xmm8, [diez]     ; xmm0 = |  10 | ... |  10 |

    .ciclo:
        movdqu xmm0, [rdi]  ; xmm0 = |  d15 | ... |  d0 |

        paddusb xmm0, xmm8  ; xmm0 = |d15+10| ... |d0+10|

        movdqu [rdi], xmm0

    add rdi, 16
    loop .ciclo
    pop rbp
    ret
```

Operaciones Aritméticas

ADDPS	ADDSS	ADDPD	ADDSD	Addition of FP values
SUBPS	SUBSS	SUBPD	SUBSD	Subtraction of FP values
MULPS	MULSS	MULPD	MULSD	Multiply of FP values
DIVPS	DIVSS	DIVPD	DIVSD	Divition of FP values
MAXPS	MAXSS	MINPS	MINSS	Max and Min of Single FP values
MAXPD	MAXSD	MINPD	MINSD	Max and Min of Double FP values

Operaciones Aritméticas

ADDPS	ADDSS	ADDPD	ADDSD	Addition of FP values
SUBPS	SUBSS	SUBPD	SUBSD	Subtraction of FP values
MULPS	MULSS	MULPD	MULSD	Multiply of FP values
DIVPS	DIVSS	DIVPD	DIVSD	Divition of FP values
MAXPS	MAXSS	MINPS	MINSS	Max and Min of Single FP values
MAXPD	MAXSD	MINPD	MINSD	Max and Min of Double FP values

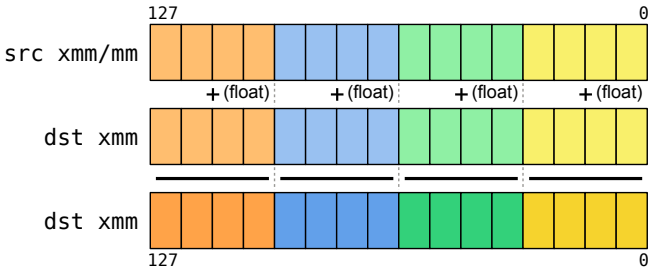
Ejemplos:

Operaciones Aritméticas

ADDPS	ADDSS	ADDPD	ADDSD	Addition of FP values
SUBPS	SUBSS	SUBPD	SUBSD	Subtraction of FP values
MULPS	MULSS	MULPD	MULSD	Multiply of FP values
DIVPS	DIVSS	DIVPD	DIVSD	Divition of FP values
MAXPS	MAXSS	MINPS	MINSS	Max and Min of Single FP values
MAXPD	MAXSD	MINPD	MINSD	Max and Min of Double FP values

Ejemplos:

```
ADDPS xmm0, [data]
```
✓

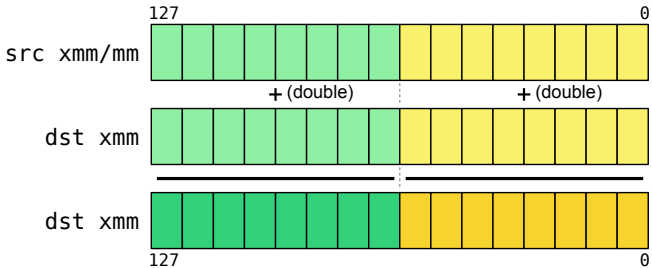


Operaciones Aritméticas

ADDPS	ADDSS	ADDPD	ADDSD	Addition of FP values
SUBPS	SUBSS	SUBPD	SUBSD	Subtraction of FP values
MULPS	MULSS	MULPD	MULSD	Multiply of FP values
DIVPS	DIVSS	DIVPD	DIVSD	Divition of FP values
MAXPS	MAXSS	MINPS	MINSS	Max and Min of Single FP values
MAXPD	MAXSD	MINPD	MINSD	Max and Min of Double FP values

Ejemplos:

```
ADDPS xmm0, [data]    ✓
ADDPD xmm0, [data]    ✓
```

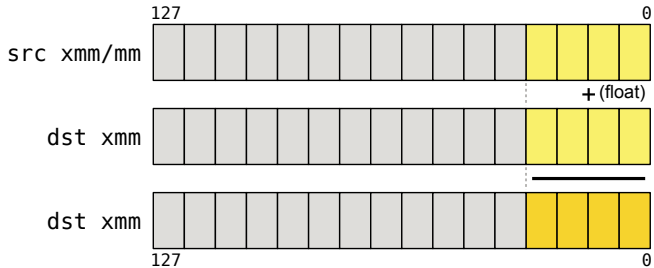


Operaciones Aritméticas

ADDPS	ADDSS	ADDPD	ADDSD	Addition of FP values
SUBPS	SUBSS	SUBPD	SUBSD	Subtraction of FP values
MULPS	MULSS	MULPD	MULSD	Multiply of FP values
DIVPS	DIVSS	DIVPD	DIVSD	Divition of FP values
MAXPS	MAXSS	MINPS	MINSS	Max and Min of Single FP values
MAXPD	MAXSD	MINPD	MINSD	Max and Min of Double FP values

Ejemplos:

```
ADDPS xmm0, [data]   ✓
ADDPD xmm0, [data]   ✓
ADDSS xmm0, [data]   ✓
```

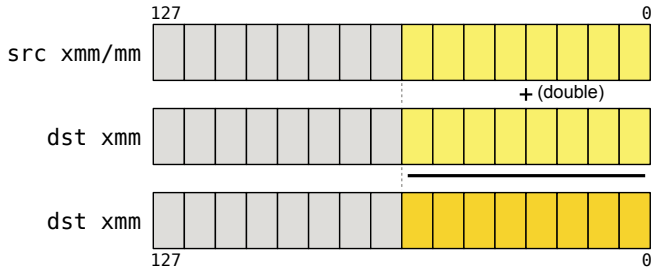


Operaciones Aritméticas

ADDPS	ADDSS	ADDPD	ADDSD	Addition of FP values
SUBPS	SUBSS	SUBPD	SUBSD	Subtraction of FP values
MULPS	MULSS	MULPD	MULSD	Multiply of FP values
DIVPS	DIVSS	DIVPD	DIVSD	Divition of FP values
MAXPS	MAXSS	MINPS	MINSS	Max and Min of Single FP values
MAXPD	MAXSD	MINPD	MINSD	Max and Min of Double FP values

Ejemplos:

```
ADDPS xmm0, [data]   ✓
ADDPD xmm0, [data]   ✓
ADDSS xmm0, [data]   ✓
ADDSD xmm0, [data]   ✓
```



Operaciones Aritméticas

ADDPS	ADDSS	ADDPD	ADDSD	Addition of FP values
SUBPS	SUBSS	SUBPD	SUBSD	Subtraction of FP values
MULPS	MULSS	MULPD	MULSD	Multiply of FP values
DIVPS	DIVSS	DIVPD	DIVSD	Divition of FP values
MAXPS	MAXSS	MINPS	MINSS	Max and Min of Single FP values
MAXPD	MAXSD	MINPD	MINSD	Max and Min of Double FP values

Ejemplos:

```
ADDPS xmm0, [data]    ✓
ADDPD xmm0, [data]    ✓
ADDSS xmm0, [data]    ✓
ADDSD xmm0, [data]    ✓
MINSD [data], xmm0    ×   Modo de direccionamiento invalido.
```

Operaciones Aritméticas

SQRTSS	SQRTPS	Square root of Scalar/Packed Single FP values
SQRTSD	SQRTPD	Square root of Scalar/Packed Double FP values

Operaciones Aritméticas

SQRTSS	SQRTPS	Square root of Scalar/Packed Single FP values
SQRTSD	SQRTPD	Square root of Scalar/Packed Double FP values

Ejemplos:

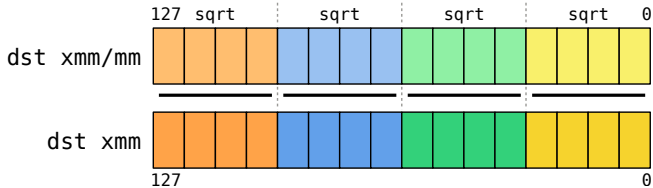
Operaciones Aritméticas

SQRTSS	SQRTPS	Square root of Scalar/Packed Single FP values
SQRTSD	SQRTPD	Square root of Scalar/Packed Double FP values

Ejemplos:

```
SQRTPS xmm0, [data]
```

✓



Operaciones Aritméticas

SQRTSS	SQRTPS	Square root of Scalar/Packed Single FP values
SQRTSD	SQRTPD	Square root of Scalar/Packed Double FP values

Ejemplos:

```
SQRTPS xmm0, [data]   ✓
SQRTSS xmm0, [data]   ✓
```

Operaciones Aritméticas

SQRTSS	SQRTPS	Square root of Scalar/Packed Single FP values
SQRTSD	SQRTPD	Square root of Scalar/Packed Double FP values

Ejemplos:

```
SQRTPS xmm0, [data]     ✓
SQRTSS xmm0, [data]     ✓
SQRTPD [data], xmm0     ×   Modo de direccionamiento invalido.
```

Ejemplo

Normalizar Vector

Dado un vector de 128 valores positivos en punto flotante de 32 bits. Normalizar los mismos y almacenar el resultado en el mismo vector.

```
void normalizar(float *vector)
```

Normalizar Vector (Parte 1/5)

```
normalizar: ; rdi = float *vector
    push rbp
    mov rbp,rsp
```

Normalizar Vector (Parte 1/5)

```
normalizar: ; rdi = float *vector
    push rbp
    mov rbp,rsp
    ; (1) find max
    mov rdx, rdi
    mov rcx, (128 >> 2)      ; rcx = 128/4
    movaps xmm1, [rdx]       ; xmm1 = | f.3 | f.2 | f.1 | f.0 |
```

Normalizar Vector (Parte 1/5)

```
normalizar: ; rdi = float *vector
    push rbp
    mov rbp,rsp
    ; (1) find max
    mov rdx, rdi
    mov rcx, (128 >> 2)      ; rcx = 128/4
    movaps xmm1, [rdx]       ; xmm1 = | f.3 | f.2 | f.1 | f.0 |
    .cicloMax:
        movaps xmm0, [rdx]   ; xmm0 = | f.i+3  | f.i+2  | f.i+1  | f.i+0  |
        maxps xmm1, xmm0     ; xmm1 = | fmax.3 | fmax.2 | fmax.1 | fmax.0 |
        add rdx, 16
    loop .cicloMax
```

Normalizar Vector (Parte 1/5)

```
normalizar: ; rdi = float *vector
    push rbp
    mov rbp,rsp
    ; (1) find max
    mov rdx, rdi
    mov rcx, (128 >> 2)      ; rcx = 128/4
    movaps xmm1, [rdx]       ; xmm1 = | f.3 | f.2 | f.1 | f.0 |
    .cicloMax:
        movaps xmm0, [rdx]   ; xmm0 = | f.i+3  | f.i+2  | f.i+1  | f.i+0  |
        maxps xmm1, xmm0     ; xmm1 = | fmax.3 | fmax.2 | fmax.1 | fmax.0 |
        add rdx, 16
    loop .cicloMax
    movdqu xmm0, xmm1  ; xmm0 = | fmax.3 | fmax.2 |  fmax.1  |  fmax.0  |
    psrldq xmm0, 8     ; xmm0 = |    0   |    0   |  fmax.3  |  fmax.2  |
    maxps xmm1, xmm0   ; xmm1 = | ...... | ...... | fmax.1y3 | fmax.0y2 |
```

Normalizar Vector (Parte 1/5)

```
normalizar: ; rdi = float *vector
    push rbp
    mov rbp,rsp
    ; (1) find max
    mov rdx, rdi
    mov rcx, (128 >> 2)      ; rcx = 128/4
    movaps xmm1, [rdx]       ; xmm1 = | f.3 | f.2 | f.1 | f.0 |
    .cicloMax:
        movaps xmm0, [rdx]   ; xmm0 = | f.i+3  | f.i+2  | f.i+1  | f.i+0  |
        maxps xmm1, xmm0     ; xmm1 = | fmax.3 | fmax.2 | fmax.1 | fmax.0 |
        add rdx, 16
    loop .cicloMax
    movdqu xmm0, xmm1  ; xmm0 = | fmax.3 | fmax.2 |  fmax.1  |  fmax.0  |
    psrldq xmm0, 8     ; xmm0 = |    0   |    0   |  fmax.3  |  fmax.2  |
    maxps xmm1, xmm0   ; xmm1 = | ...... | ...... | fmax.1y3 | fmax.0y2 |
    movdqu xmm0, xmm1  ; xmm0 = | ...... | ...... | fmax.1y3 | fmax.0y2 |
    psrldq xmm0, 4     ; xmm0 = |    0   | ...... | ........ | fmax.1y3 |
    maxps xmm1, xmm0   ; xmm1 = | ...... | ...... | ........ |   fmax   |
```

Normalizar Vector (Parte 2/5)

```
normalizar: ; rdi = float *vector
    ...


    ; (2) broadcast max
    pslldq xmm1, 12   ; xmm1 = | max |  0  |  0  |  0  |
    movdqu xmm0, xmm1 ; xmm0 = | max |  0  |  0  |  0  |
```

Normalizar Vector (Parte 2/5)

```
normalizar: ; rdi = float *vector
    ...


    ; (2) broadcast max
    pslldq xmm1, 12   ; xmm1 = | max |  0  |  0  |  0  |
    movdqu xmm0, xmm1 ; xmm0 = | max |  0  |  0  |  0  |

    psrldq xmm1, 4    ; xmm1 = |  0  | max |  0  |  0  |
    por    xmm1, xmm0 ; xmm1 = | max | max |  0  |  0  |
```

Normalizar Vector (Parte 2/5)

```
normalizar: ; rdi = float *vector
    ...


    ; (2) broadcast max
    pslldq xmm1, 12   ; xmm1 = | max |  0  |  0  |  0  |
    movdqu xmm0, xmm1 ; xmm0 = | max |  0  |  0  |  0  |

    psrldq xmm1, 4    ; xmm1 = |  0  | max |  0  |  0  |
    por    xmm1, xmm0 ; xmm1 = | max | max |  0  |  0  |

    movdqu xmm0, xmm1 ; xmm0 = | max | max |  0  |  0  |
    psrldq xmm1, 8    ; xmm1 = |  0  |  0  | max | max |
    por    xmm1, xmm0 ; xmm1 = | max | max | max | max |
```

Normalizar Vector (Parte 3/5)

```
normalizar: ; rdi = float *vector
    ...
```

Normalizar Vector (Parte 3/5)

```
normalizar: ; rdi = float *vector
    ...

    ; (3) find min
    mov rdx, rdi
    mov rcx, (128 >> 2)       ; rcx = 128/4
    movaps xmm2, [rdx]        ; xmm1 = | f.3 | f.2 | f.1 | f.0 |
    .cicloMin:
        movaps xmm0, [rdx]    ; xmm0 = | f.i+3  | f.i+2  | f.i+1  | f.i+0  |
        minps xmm2, xmm0      ; xmm1 = | fmin.3 | fmin.2 | fmin.1 | fmin.0 |
        add rdx, 16
    loop .cicloMin
```

Normalizar Vector (Parte 3/5)

```
normalizar: ; rdi = float *vector
    ...

    ; (3) find min
    mov rdx, rdi
    mov rcx, (128 >> 2)       ; rcx = 128/4
    movaps xmm2, [rdx]        ; xmm1 = | f.3 | f.2 | f.1 | f.0 |
    .cicloMin:
        movaps xmm0, [rdx]    ; xmm0 = | f.i+3  | f.i+2  | f.i+1  | f.i+0  |
        minps xmm2, xmm0      ; xmm1 = | fmin.3 | fmin.2 | fmin.1 | fmin.0 |
        add rdx, 16
    loop .cicloMin
    movdqu xmm0, xmm2  ; xmm0 = | fmin.3 | fmin.2 |  fmin.1  |  fmin.0  |
    psrldq xmm0, 8     ; xmm0 = |    0   |    0   |  fmin.3  |  fmin.2  |
    minps xmm2, xmm0   ; xmm2 = | ...... | ...... | fmin.1y3 | fmin.0y2 |
    movdqu xmm0, xmm2  ; xmm0 = | ...... | ...... | fmin.1y3 | fmin.0y2 |
    psrldq xmm0, 4     ; xmm0 = |    0   | ...... | ........ | fmin.1y3 |
    minps xmm2, xmm0   ; xmm2 = | ...... | ...... | ........ |   fmin   |
```

Normalizar Vector (Parte 4/5)

```
normalizar: ; rdi = float *vector
    ...


    ; (4) broadcast min
    pslldq xmm2, 12   ; xmm2 = | min |  0  |  0  |  0  |
    movdqu xmm0, xmm2 ; xmm0 = | min |  0  |  0  |  0  |

    psrldq xmm2, 4    ; xmm2 = |  0  | min |  0  |  0  |
    por    xmm2, xmm0 ; xmm2 = | min | min |  0  |  0  |

    movdqu xmm0, xmm2 ; xmm0 = | min | min |  0  |  0  |
    psrldq xmm2, 8    ; xmm2 = |  0  |  0  | min | min |
    por    xmm2, xmm0 ; xmm2 = | min | min | min | min |
```

Normalizar Vector (Parte 5/5)

```
normalizar: ; rdi = float *vector
    ...

    ; (5) normalizacion
    subps xmm1, xmm2     ; xmm1 = | max-min | max-min | max-min | max-min |
    mov rdx, rdi         ; rdx = vector
    mov rcx, (128 >> 2)  ; rcx = 128/4
```

Normalizar Vector (Parte 5/5)

```
normalizar: ; rdi = float *vector
    ...

    ; (5) normalizacion
    subps xmm1, xmm2     ; xmm1 = | max-min | max-min | max-min | max-min |
    mov rdx, rdi         ; rdx = vector
    mov rcx, (128 >> 2)  ; rcx = 128/4

    .ciclo:
        movaps xmm0, [rdx]
        divps xmm0, xmm1  ; xmm0 = | f.i+3/(max-min) | ... | f.i+0/(max-min) |
        movaps [rdx], xmm0
        add rdx, 16
    loop .ciclo
```

Normalizar Vector (Parte 5/5)

```
normalizar: ; rdi = float *vector
    ...

    ; (5) normalizacion
    subps xmm1, xmm2     ; xmm1 = | max-min | max-min | max-min | max-min |
    mov rdx, rdi         ; rdx = vector
    mov rcx, (128 >> 2)  ; rcx = 128/4

    .ciclo:
        movaps xmm0, [rdx]
        divps xmm0, xmm1  ; xmm0 = | f.i+3/(max-min) | ... | f.i+0/(max-min) |
        movaps [rdx], xmm0
        add rdx, 16
    loop .ciclo

    pop rbp
    ret
```

Normalizar Vector

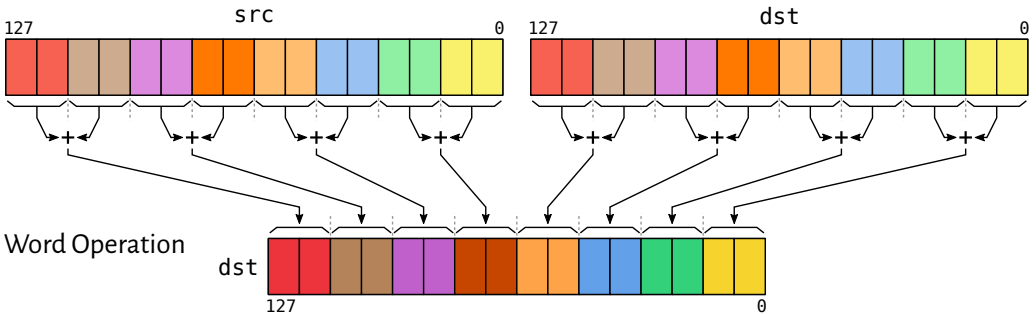
```
normalizar: ; rdi = float *vector
    push rbp
    mov rbp,rsp
    ; (1) find max
    mov rdx, rdi
    mov rcx, (128 >> 2)
    movaps xmm1, [rdx]
    .cicloMax:
        movaps xmm0, [rdx]
        maxps xmm1, xmm0
        add rdx, 16
    loop .cicloMax
    movdqu xmm0, xmm1
    psrldq xmm0, 8
    maxps xmm1, xmm0
    movdqu xmm0, xmm1
    psrldq xmm0, 4
    maxps xmm1, xmm0
    ; (2) broadcast max
    pslldq xmm1, 12   ; xmm1 = |AA|00|00|00|
    movdqu xmm0, xmm1 ; xmm0 = |AA|00|00|00|
    psrldq xmm1, 4    ; xmm1 = |00|AA|00|00|
    por    xmm1, xmm0 ; xmm1 = |AA|AA|00|00|
    movdqu xmm0, xmm1 ; xmm0 = |AA|AA|00|00|
    psrldq xmm1, 8    ; xmm1 = |00|00|AA|AA|
    por    xmm1, xmm0 ; xmm1 = |AA|AA|AA|AA|
    ; (3) find min
    mov rdx, rdi
    mov rcx, (128 >> 2)
    movaps xmm2, [rdx]
    .cicloMin:
        movaps xmm0, [rdx]
        minps xmm2, xmm0
        add rdx, 16
    loop .cicloMin
    movdqu xmm0, xmm2
    psrldq xmm0, 8
    minps xmm2, xmm0
    movdqu xmm0, xmm2
    psrldq xmm0, 4
    minps xmm2, xmm0
    ; (4) broadcast min
    pslldq xmm2, 12   ; xmm2 = |AA|00|00|00|
    movdqu xmm0, xmm2 ; xmm0 = |AA|00|00|00|
    psrldq xmm2, 4    ; xmm2 = |00|AA|00|00|
    por    xmm2, xmm0 ; xmm2 = |AA|AA|00|00|
    movdqu xmm0, xmm2 ; xmm0 = |AA|AA|00|00|
    psrldq xmm2, 8    ; xmm2 = |00|00|AA|AA|
    por    xmm2, xmm0 ; xmm2 = |AA|AA|AA|AA|
    ; (5) normalizacion
    subps xmm1, xmm2
    mov rdx, rdi
    mov rcx, (128 >> 2)
    .ciclo:
    movaps xmm0, [rdx]
    divps xmm0, xmm1
    movaps [rdx], xmm0
    add rdx, 16
    loop .ciclo
    pop rbp
    ret
```

Operaciones Aritméticas

PHADDW	PHADDD	Horizontal addition of unsigned 16bit/32bit integers
PHADDSW		Horizontal saturated addition of 16bit integers
PHSUBW	PHSUBD	Horizontal subtraction of unsigned 16bit/32bit integers
PHSUBSW		Horizontal saturated subtraction of 16bit words
HADDPS	HADDPD	Packed Single/Double FP Horizontal Add
HSUBPS	HSUBPD	Packed Single/Double FP Horizontal Subtract

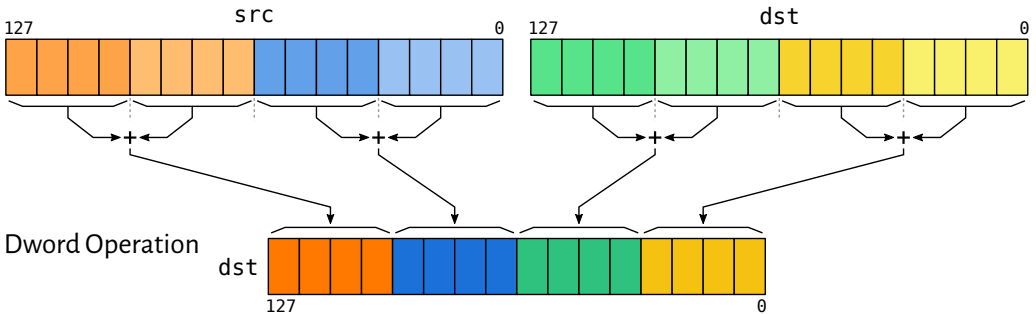
Operaciones Aritméticas

PHADDW	PHADDD	Horizontal addition of unsigned 16bit/32bit integers
PHADDSW		Horizontal saturated addition of 16bit integers
PHSUBW	PHSUBD	Horizontal subtraction of unsigned 16bit/32bit integers
PHSUBSW		Horizontal saturated subtraction of 16bit words
HADDPS	HADDPD	Packed Single/Double FP Horizontal Add
HSUBPS	HSUBPD	Packed Single/Double FP Horizontal Subtract



Operaciones Aritméticas

PHADDW	PHADDD	Horizontal addition of unsigned 16bit/32bit integers
PHADDSW		Horizontal saturated addition of 16bit integers
PHSUBW	PHSUBD	Horizontal subtraction of unsigned 16bit/32bit integers
PHSUBSW		Horizontal saturated subtraction of 16bit words
HADDPS	HADDPD	Packed Single/Double FP Horizontal Add
HSUBPS	HSUBPD	Packed Single/Double FP Horizontal Subtract



Operaciones Lógicas

PAND	PANDN	POR	PXOR	Operaciones lógicas para enteros.
ANDPS	ANDNPS	ORPS	XORPS	Operaciones lógicas para *float*.
ANDPD	ANDNPD	ORPD	XORPD	Operaciones lógicas para *double*.

- Actuan lógicamente sobre todo el registro, sin importa el tamaño del operando.
- La distinción entre PS y PD se debe a meta información para el procesador.

Operaciones Lógicas

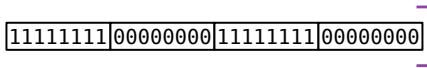
PAND	PANDN	POR	PXOR	Operaciones lógicas para enteros.
ANDPS	ANDNPS	ORPS	XORPS	Operaciones lógicas para *float*.
ANDPD	ANDNPD	ORPD	XORPD	Operaciones lógicas para *double*.

- Actuan lógicamente sobre todo el registro, sin importa el tamaño del operando.
- La distinción entre PS y PD se debe a meta información para el procesador.

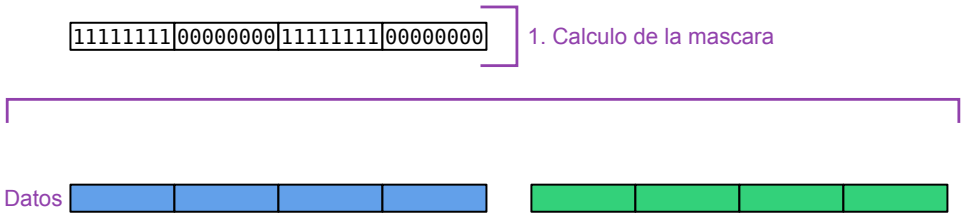
PSLLW	PSLLD	PSLLQ	PSLLDQ*
PSRLW	PSRLD	PSRLQ	PSRLDQ*
PSRAW	PSRAD		

- Todos los *shifts* operan de forma lógica como aritmética, tanto a derecha como izquierda.
- Se limitan a realizar la operación sobre cada uno de los datos dentro del registro según su tamaño.
- * En las operaciones indicas, el parámetro es la cantidad de bytes del desplazamiento.

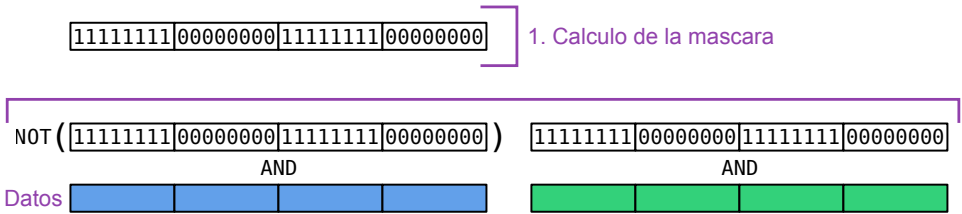
Técnica: Operatoria con mascaras



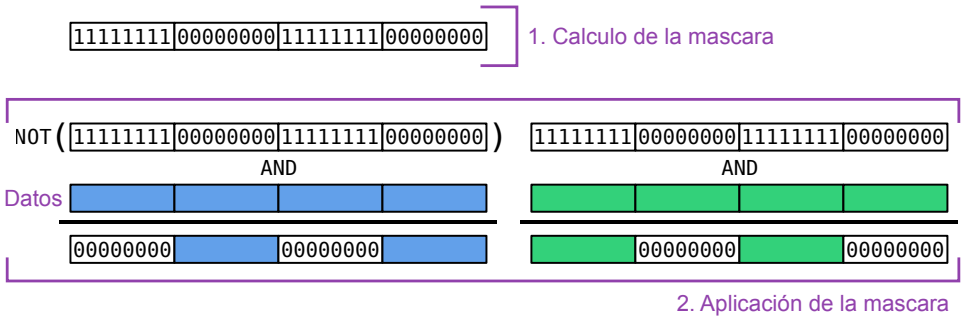
Técnica: Operatoria con mascaras



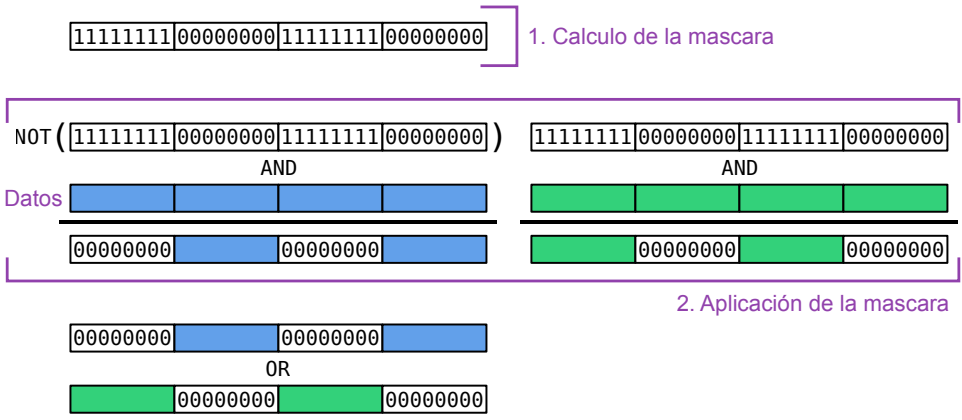
Técnica: Operatoria con mascaras



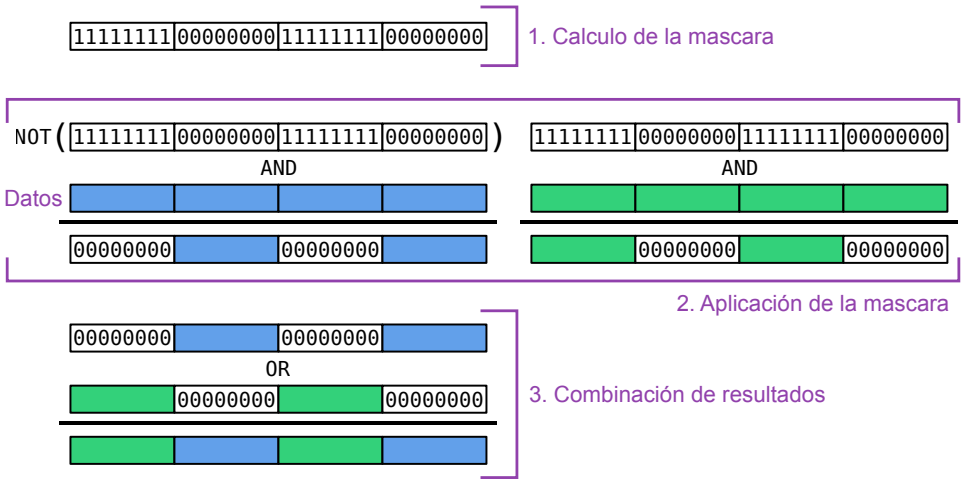
Técnica: Operatoria con mascaras



Técnica: Operatoria con mascaras



Técnica: Operatoria con mascaras



Operaciones de comparación

PCMPEQB	PCMPEQW	PCMPEQD	PCMPEQQ	Compare Packed Data for Equal
PCMPGTB	PCMPGTW	PCMPGTD	PCMPGTQ	Compare Packed Signed Int for Greater Than

Operaciones de comparación

PCMPEQB	PCMPEQW	PCMPEQD	PCMPEQQ	Compare Packed Data for Equal
PCMPGTB	PCMPGTW	PCMPGTD	PCMPGTQ	Compare Packed Signed Int for Greater Than

Ejemplos:

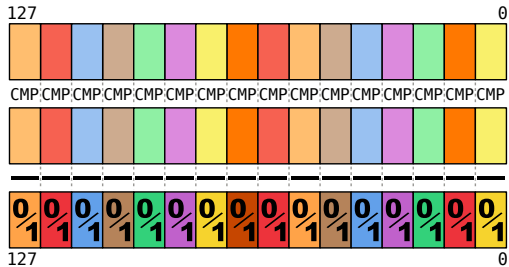
Operaciones de comparación

PCMPEQB	PCMPEQW	PCMPEQD	PCMPEQQ	Compare Packed Data for Equal
PCMPGTB	PCMPGTW	PCMPGTD	PCMPGTQ	Compare Packed Signed Int for Greater Than

Ejemplos:

```
PCMPEQB xmm0, [data]
```

✓

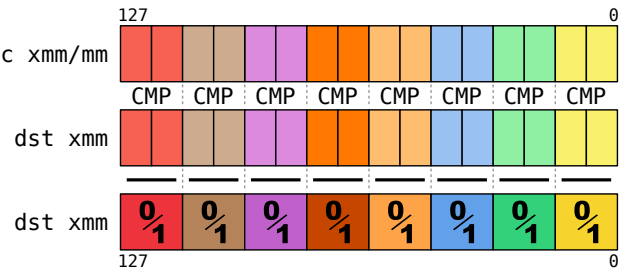


Operaciones de comparación

PCMPEQB	PCMPEQW	PCMPEQD	PCMPEQQ	Compare Packed Data for Equal
PCMPGTB	PCMPGTW	PCMPGTD	PCMPGTQ	Compare Packed Signed Int for Greater Than

Ejemplos:

```
PCMPEQB xmm0, [data]   ✓
PCMPEQW xmm0, [data]   ✓
```

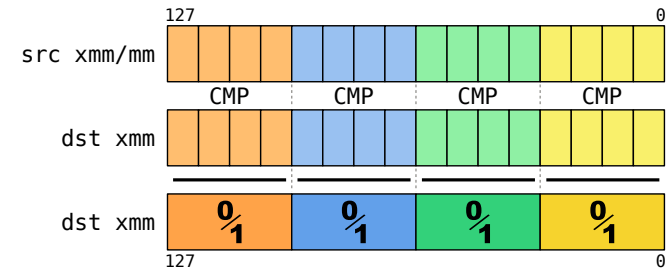


Operaciones de comparación

PCMPEQB	PCMPEQW	PCMPEQD	PCMPEQQ	Compare Packed Data for Equal
PCMPGTB	PCMPGTW	PCMPGTD	PCMPGTQ	Compare Packed Signed Int for Greater Than

Ejemplos:

```
PCMPEQB xmm0, [data]   ✓
PCMPEQW xmm0, [data]   ✓
PCMPEQD xmm0, [data]   ✓
```

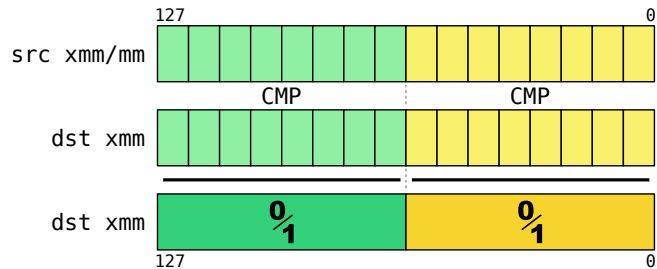


Operaciones de comparación

PCMPEQB	PCMPEQW	PCMPEQD	PCMPEQQ	Compare Packed Data for Equal
PCMPGTB	PCMPGTW	PCMPGTD	PCMPGTQ	Compare Packed Signed Int for Greater Than

Ejemplos:

```
PCMPEQB xmm0, [data]   ✓
PCMPEQW xmm0, [data]   ✓
PCMPEQD xmm0, [data]   ✓
PCMPEQQ xmm0, [data]   ✓
```



Operaciones de comparación

PCMPEQB	PCMPEQW	PCMPEQD	PCMPEQQ	Compare Packed Data for Equal
PCMPGTB	PCMPGTW	PCMPGTD	PCMPGTQ	Compare Packed Signed Int for Greater Than

Ejemplos:

```
PCMPEQB xmm0, [data]    ✓
PCMPEQW xmm0, [data]    ✓
PCMPEQD xmm0, [data]    ✓
PCMPEQQ xmm0, [data]    ✓
PCMPGTQ [data], xmm0    ×   Modo de direccionamiento invalido.
```

Operaciones de comparación

CMPxxPD	Compare Packed Double-Precision Floating-Point Values
CMPxxPS	Compare Packed Single-Precision Floating-Point Values
CMPxxSD	Compare Scalar Double-Precision Floating-Point Values
CMPxxSS	Compare Scalar Single-Precision Floating-Point Values
COMISD	Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS
COMISS	Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

	Acción	xx	CMPxxyy A, B
0	Igual	EQ	$A = B$
1	Menor	LT	$A < B$
2	Menor o Igual	LE	$A \leqslant B$
3	No Orden	UNORD	$A, B = unordered$
4	Distinto	NEQ	$A \neq B$
5	No Menor	NLT	$not(A < B)$
6	No Meno o Igual	NLE	$not(A \leqslant B)$
7	Orden	ORD	$A, B = Ordered$

Operaciones de comparación

CMPxxPD	Compare Packed Double-Precision Floating-Point Values
CMPxxPS	Compare Packed Single-Precision Floating-Point Values
CMPxxSD	Compare Scalar Double-Precision Floating-Point Values
CMPxxSS	Compare Scalar Single-Precision Floating-Point Values
COMISD	Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS
COMISS	Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

Ejemplos:

	Acción	xx	CMPxxyy A, B
0	Igual	EQ	$A = B$
1	Menor	LT	$A < B$
2	Menor o Igual	LE	$A \leqslant B$
3	No Orden	UNORD	$A, B = unordered$
4	Distinto	NEQ	$A \neq B$
5	No Menor	NLT	$not(A < B)$
6	No Meno o Igual	NLE	$not(A \leqslant B)$
7	Orden	ORD	$A, B = Ordered$

Operaciones de comparación

CMPxxPD	Compare Packed Double-Precision Floating-Point Values
CMPxxPS	Compare Packed Single-Precision Floating-Point Values
CMPxxSD	Compare Scalar Double-Precision Floating-Point Values
CMPxxSS	Compare Scalar Single-Precision Floating-Point Values
COMISD	Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS
COMISS	Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

Ejemplos:

```
CMPEQPD xmm0, [data]
```

✓

	Acción	xx	CMPxxyy A, B
0	Igual	EQ	$A = B$
1	Menor	LT	$A < B$
2	Menor o Igual	LE	$A \leqslant B$
3	No Orden	UNORD	$A, B = unordered$
4	Distinto	NEQ	$A \neq B$
5	No Menor	NLT	$not(A < B)$
6	No Meno o Igual	NLE	$not(A \leqslant B)$
7	Orden	ORD	$A, B = Ordered$

Operaciones de comparación

CMPxxPD	Compare Packed Double-Precision Floating-Point Values
CMPxxPS	Compare Packed Single-Precision Floating-Point Values
CMPxxSD	Compare Scalar Double-Precision Floating-Point Values
CMPxxSS	Compare Scalar Single-Precision Floating-Point Values
COMISD	Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS
COMISS	Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

Ejemplos:

```
CMPEQPD xmm0, [data]    ✓
CMPLEPD xmm0, [data]    ✓
```

	Acción	xx	CMPxxyy A, B
0	Igual	EQ	$A = B$
1	Menor	LT	$A < B$
2	Menor o Igual	LE	$A \leqslant B$
3	No Orden	UNORD	$A, B = unordered$
4	Distinto	NEQ	$A \neq B$
5	No Menor	NLT	$not(A < B)$
6	No Meno o Igual	NLE	$not(A \leqslant B)$
7	Orden	ORD	$A, B = Ordered$

Operaciones de comparación

CMPxxPD	Compare Packed Double-Precision Floating-Point Values
CMPxxPS	Compare Packed Single-Precision Floating-Point Values
CMPxxSD	Compare Scalar Double-Precision Floating-Point Values
CMPxxSS	Compare Scalar Single-Precision Floating-Point Values
COMISD	Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS
COMISS	Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

Ejemplos:

```
CMPEQPD xmm0, [data]      ✓
CMPLEPD xmm0, [data]      ✓
CMPORDPD xmm0, [data]     ✓    ; (Nan)
```

	Acción	xx	CMPxxyy A, B
0	Igual	EQ	$A = B$
1	Menor	LT	$A < B$
2	Menor o Igual	LE	$A \leqslant B$
3	No Orden	UNORD	$A, B = unordered$
4	Distinto	NEQ	$A \neq B$
5	No Menor	NLT	$not(A < B)$
6	No Meno o Igual	NLE	$not(A \leqslant B)$
7	Orden	ORD	$A, B = Ordered$

Ejemplo

Suma pares

Dado un vector de 128 enteros con signo de 16 bits. Sumar todos los valores pares y retornar el resultado de la suma en 32 bits.

```
int32_t sumarPares(int16_t *v)
```

Suma pares

```
sumarpares: ; rdi = int16_t *v
    push rbp
    mov rbp,rsp
```

Suma pares

```
sumarpares: ; rdi = int16_t *v
    push rbp
    mov rbp,rsp
    mov rcx, (128 >> 2)  ; rcx = 128 / 4
    pxor xmm8, xmm8      ; xmm8 = | 0 | 0 | 0 | 0 |
```

Suma pares

```
sumarpares: ; rdi = int16_t *v
    push rbp
    mov rbp,rsp
    mov rcx, (128 >> 2)  ; rcx = 128 / 4
    pxor xmm8, xmm8      ; xmm8 = | 0 | 0 | 0 | 0 |
    .ciclo:
        pmovsxwd xmm0, [rdi] ; (ejemplo) xmm0 = | 00001233 | 00007314 | 00003011 | FFFF9311 |
```

Suma pares

```
sumarpares: ; rdi = int16_t *v
    push rbp
    mov rbp,rsp
    mov rcx, (128 >> 2)  ; rcx = 128 / 4
    pxor xmm8, xmm8      ; xmm8 = | 0 | 0 | 0 | 0 |
    .ciclo:
        pmovsxwd xmm0, [rdi] ; (ejemplo) xmm0 = | 00001233 | 00007314 | 00003011 | FFFF9311 |
        pabsd xmm1, xmm0     ; (ejemplo) xmm1 = | 00001233 | 00007314 | 00003011 | 00006CEE |
```

Suma pares

```
sumarpares: ; rdi = int16_t *v
    push rbp
    mov rbp,rsp
    mov rcx, (128 >> 2)  ; rcx = 128 / 4
    pxor xmm8, xmm8      ; xmm8 = | 0 | 0 | 0 | 0 |
    .ciclo:
        pmovsxwd xmm0, [rdi] ; (ejemplo) xmm0 = | 00001233 | 00007314 | 00003011 | FFFF9311 |
        pabsd xmm1, xmm0     ; (ejemplo) xmm1 = | 00001233 | 00007314 | 00003011 | 00006CEE |
        pslld  xmm1, 31      ; (ejemplo) xmm1 = | 80000000 | 00000000 | 80000000 | 00000000 |
```

Suma pares

```
sumarpares: ; rdi = int16_t *v
    push rbp
    mov rbp,rsp
    mov rcx, (128 >> 2)  ; rcx = 128 / 4
    pxor xmm8, xmm8      ; xmm8 = | 0 | 0 | 0 | 0 |
    .ciclo:
        pmovsxwd xmm0, [rdi] ; (ejemplo) xmm0 = | 00001233 | 00007314 | 00003011 | FFFF9311 |

        pabsd xmm1, xmm0     ; (ejemplo) xmm1 = | 00001233 | 00007314 | 00003011 | 00006CEE |
        pslld  xmm1, 31      ; (ejemplo) xmm1 = | 80000000 | 00000000 | 80000000 | 00000000 |
        psrad  xmm1, 31      ; (ejemplo) xmm1 = | FFFFFFFF | 00000000 | FFFFFFFF | 00000000 |
```

Suma pares

```
sumarpares: ; rdi = int16_t *v
    push rbp
    mov rbp,rsp
    mov rcx, (128 >> 2)  ; rcx = 128 / 4
    pxor xmm8, xmm8      ; xmm8 = | 0 | 0 | 0 | 0 |
    .ciclo:
        pmovsxwd xmm0, [rdi] ; (ejemplo) xmm0 = | 00001233 | 00007314 | 00003011 | FFFF9311 |

        pabsd xmm1, xmm0     ; (ejemplo) xmm1 = | 00001233 | 00007314 | 00003011 | 00006CEE |
        pslld  xmm1, 31      ; (ejemplo) xmm1 = | 80000000 | 00000000 | 80000000 | 00000000 |
        psrad  xmm1, 31      ; (ejemplo) xmm1 = | FFFFFFFF | 00000000 | FFFFFFFF | 00000000 |
        pandn  xmm1, xmm0    ; (ejemplo) xmm1 = | 00000000 | 00007314 | 00000000 | FFFF9311 |
```

Suma pares

```
sumarpares: ; rdi = int16_t *v
    push rbp
    mov rbp,rsp
    mov rcx, (128 >> 2)  ; rcx = 128 / 4
    pxor xmm8, xmm8      ; xmm8 = | 0 | 0 | 0 | 0 |
    .ciclo:
        pmovsxwd xmm0, [rdi] ; (ejemplo) xmm0 = | 00001233 | 00007314 | 00003011 | FFFF9311 |

        pabsd xmm1, xmm0     ; (ejemplo) xmm1 = | 00001233 | 00007314 | 00003011 | 00006CEE |
        pslld  xmm1, 31      ; (ejemplo) xmm1 = | 80000000 | 00000000 | 80000000 | 00000000 |
        psrad  xmm1, 31      ; (ejemplo) xmm1 = | FFFFFFFF | 00000000 | FFFFFFFF | 00000000 |
        pandn  xmm1, xmm0    ; (ejemplo) xmm1 = | 00000000 | 00007314 | 00000000 | FFFF9311 |

        paddd  xmm8, xmm1    ;           xmm8 = |   SUM3   |   SUM2   |   SUM1   |   SUM0   |
        add rdi, 8
    loop .ciclo
```

Suma pares

```
sumarpares: ; rdi = int16_t *v
    push rbp
    mov rbp,rsp
    mov rcx, (128 >> 2)  ; rcx = 128 / 4
    pxor xmm8, xmm8      ; xmm8 = | 0 | 0 | 0 | 0 |
    .ciclo:
        pmovsxwd xmm0, [rdi] ; (ejemplo) xmm0 = | 00001233 | 00007314 | 00003011 | FFFF9311 |

        pabsd xmm1, xmm0     ; (ejemplo) xmm1 = | 00001233 | 00007314 | 00003011 | 00006CEE |
        pslld  xmm1, 31      ; (ejemplo) xmm1 = | 80000000 | 00000000 | 80000000 | 00000000 |
        psrad  xmm1, 31      ; (ejemplo) xmm1 = | FFFFFFFF | 00000000 | FFFFFFFF | 00000000 |
        pandn  xmm1, xmm0    ; (ejemplo) xmm1 = | 00000000 | 00007314 | 00000000 | FFFF9311 |

        paddd  xmm8, xmm1    ;           xmm8 = |   SUM3   |   SUM2   |   SUM1   |   SUM0   |
        add rdi, 8
    loop .ciclo
    phaddd xmm8, xmm8  ; xmm8 = | ... | ... | SUM3+SUM2 |      SUM1+SUM0      |
    phaddd xmm8, xmm8  ; xmm8 = | ... | ... |    ...    | SUM3+SUM2+SUM1+SUM0 |
```

Suma pares

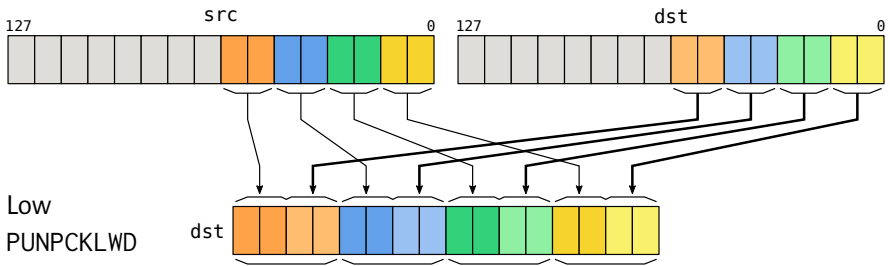
```
sumarpares: ; rdi = int16_t *v
    push rbp
    mov rbp,rsp
    mov rcx, (128 >> 2)  ; rcx = 128 / 4
    pxor xmm8, xmm8      ; xmm8 = | 0 | 0 | 0 | 0 |
    .ciclo:
        pmovsxwd xmm0, [rdi] ; (ejemplo) xmm0 = | 00001233 | 00007314 | 00003011 | FFFF9311 |

        pabsd xmm1, xmm0     ; (ejemplo) xmm1 = | 00001233 | 00007314 | 00003011 | 00006CEE |
        pslld  xmm1, 31      ; (ejemplo) xmm1 = | 80000000 | 00000000 | 80000000 | 00000000 |
        psrad  xmm1, 31      ; (ejemplo) xmm1 = | FFFFFFFF | 00000000 | FFFFFFFF | 00000000 |
        pandn  xmm1, xmm0    ; (ejemplo) xmm1 = | 00000000 | 00007314 | 00000000 | FFFF9311 |

        paddd  xmm8, xmm1    ;           xmm8 = |   SUM3   |   SUM2   |   SUM1   |   SUM0   |
        add rdi, 8
    loop .ciclo
    phaddd xmm8, xmm8  ; xmm8 = | ... | ... | SUM3+SUM2 |      SUM1+SUM0      |
    phaddd xmm8, xmm8  ; xmm8 = | ... | ... |    ...    | SUM3+SUM2+SUM1+SUM0 |
    movd eax, xmm8     ; eax = SUM3+SUM2+SUM1+SUM0
    pop rbp
    ret
```

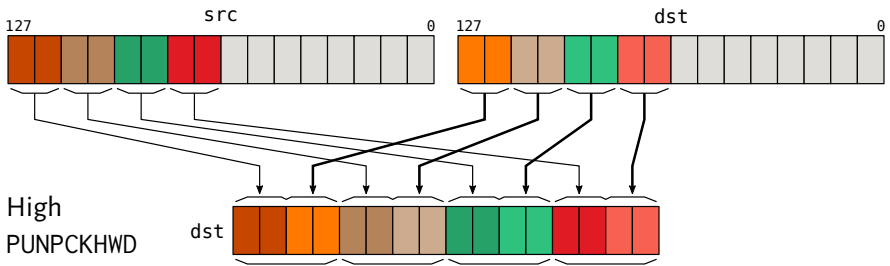
Operaciones de desempaquetado (Unpack)

PUNPCKLBW	PUNPCKHBW	Unpacks 8 enteros de 8 bits en words
PUNPCKLWD	PUNPCKHWD	Unpacks 4 enteros de 16 bits en dwords
PUNPCKLDQ	PUNPCKHDQ	Unpacks 2 enteros de 32 bits en qwords
PUNPCKLQDQ	PUNPCKHQDQ	Unpacks 1 entero de 64 bits en 128 bits
UNPCKLPS	UNPCKHPS	Unpacks Single FP
UNPCKLPD	UNPCKHPD	Unpacks Double FP



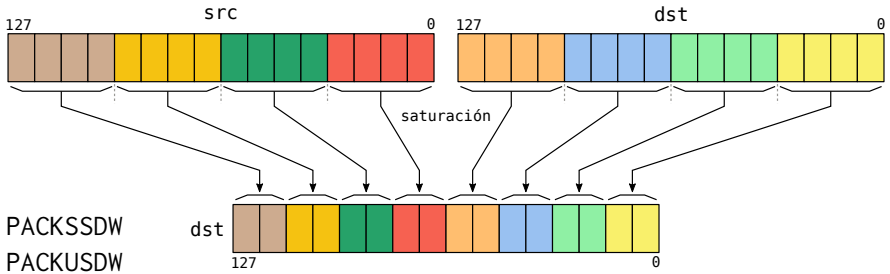
Operaciones de desempaquetado (Unpack)

PUNPCKLBW	PUNPCKHBW	Unpacks 8 enteros de 8 bits en words
PUNPCKLWD	PUNPCKHWD	Unpacks 4 enteros de 16 bits en dwords
PUNPCKLDQ	PUNPCKHDQ	Unpacks 2 enteros de 32 bits en qwords
PUNPCKLQDQ	PUNPCKHQDQ	Unpacks 1 entero de 64 bits en 128 bits
UNPCKLPS	UNPCKHPS	Unpacks Single FP
UNPCKLPD	UNPCKHPD	Unpacks Double FP



Operaciones de desempaquetado (Unpack)

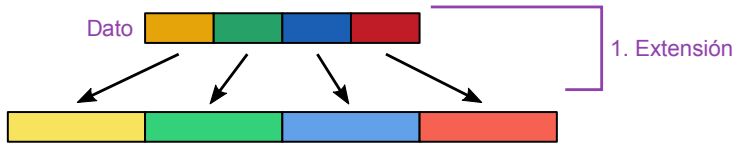
PACKSSDW	Packs 32 bits (signado) a 16 bits (signado) usando saturation
PACKUSDW	Packs 32 bits (signado) a 16 bits (sin signo) usando saturation
PACKSSWB	Packs 16 bits (signado) a 8 bits (signado) usando saturation
PACKUSWB	Packs 16 bits (signado) a 8 bits (sin signo) usando saturation



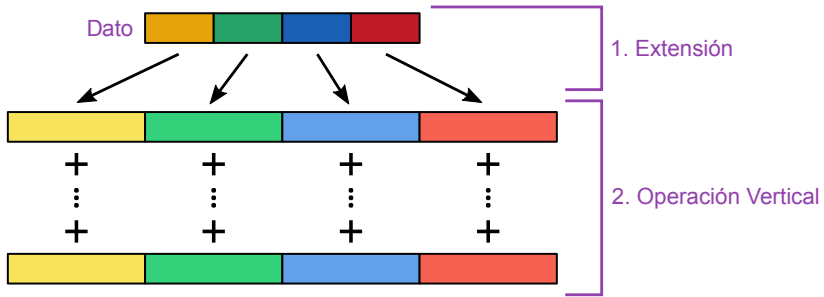
Técnica: Operatoria de desempaquetado y empaquetado

Técnica: Operatoria de desempaquetado y empaquetado

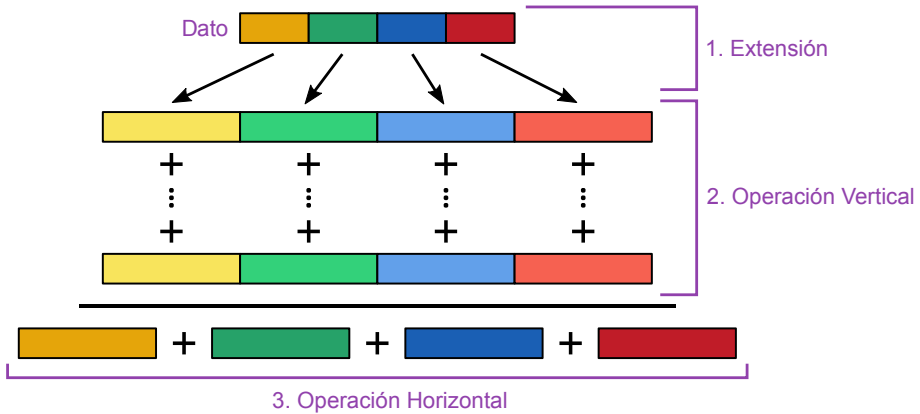
Técnica: Operatoria de desempaquetado y empaquetado



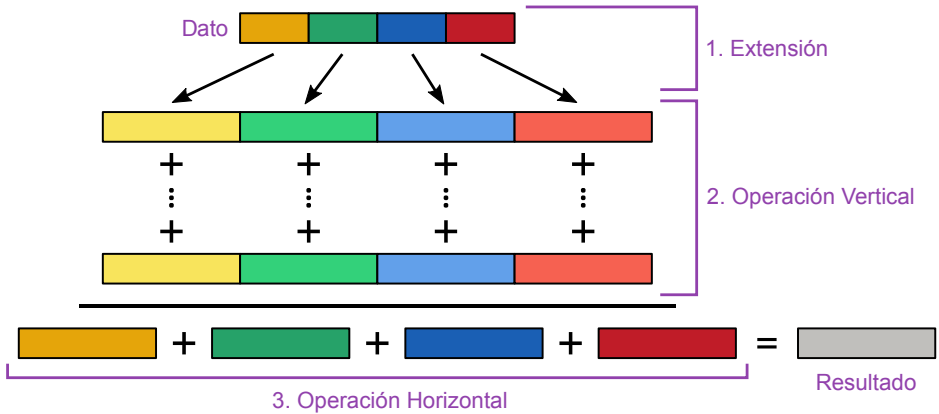
Técnica: Operatoria de desempaquetado y empaquetado



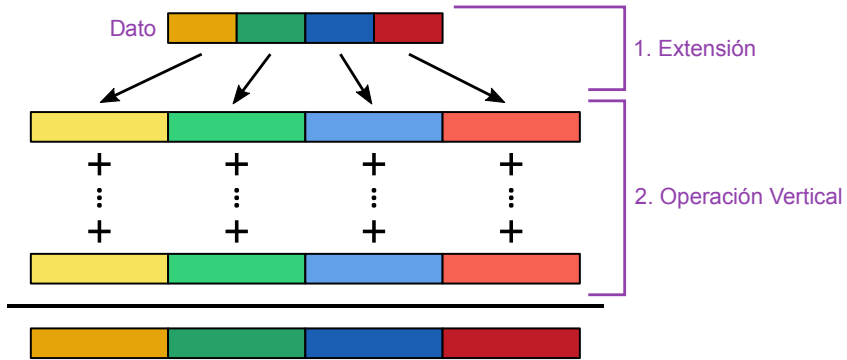
Técnica: Operatoria de desempaquetado y empaquetado



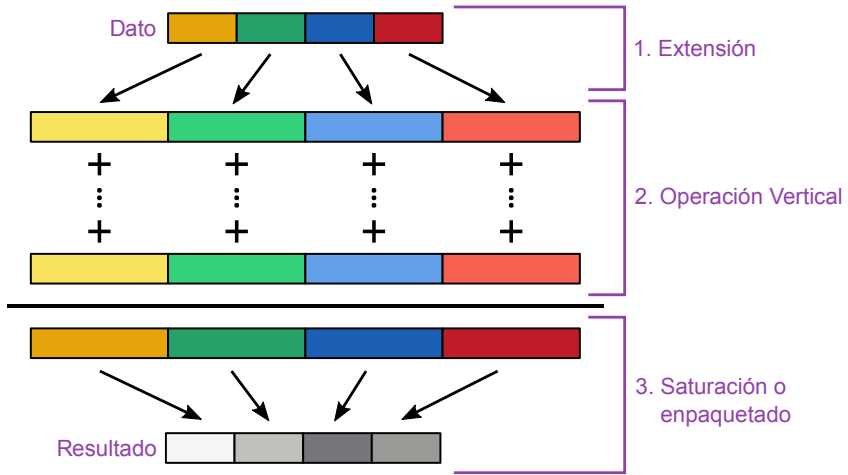
Técnica: Operatoria de desempaquetado y empaquetado



Técnica: Operatoria de desempaquetado y empaquetado



Técnica: Operatoria de desempaquetado y empaquetado



Ejemplo

Multiplicar vectores

Dado dos vectores de 128 enteros con signo de 16 bits. Multiplicar cada uno de ellos entre si y almacenar el resultado en un vector de enteros de 32 bits.

```
void mulvec(int16_t *v1, int16_t *v2, int32_t *resultado)
```

Multiplicar vectores

```
mulvec: ; rdi = int16_t *v1, rsi = int16_t *v2, rdx = int32_t *resultado
push rbp
mov rbp,rsp
mov rcx, (128 >> 2) ; rcx = 128 / 8
```

Multiplicar vectores

```
mulvec: ; rdi = int16_t *v1, rsi = int16_t *v2, rdx = int32_t *resultado
push rbp
mov rbp,rsp
mov rcx, (128 >> 2) ; rcx = 128 / 8
.ciclo:
    movdqa xmm0, [rdi]    ; xmm0 = | a7 | a6 | a5 | a4 | a3 | a2 | a1 | a0 |
    movdqa xmm1, [rsi]    ; xmm1 = | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
```

Multiplicar vectores

```
mulvec: ; rdi = int16_t *v1, rsi = int16_t *v2, rdx = int32_t *resultado
push rbp
mov rbp,rsp
mov rcx, (128 >> 2) ; rcx = 128 / 8
.ciclo:
    movdqa xmm0, [rdi]    ; xmm0 = | a7 | a6 | a5 | a4 | a3 | a2 | a1 | a0 |
    movdqa xmm1, [rsi]    ; xmm1 = | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
    movdqa xmm2, xmm0     ; xmm2 = | a7 | a6 | a5 | a4 | a3 | a2 | a1 | a0 |
    pmulhw xmm2, xmm1     ; xmm2 = | hi(a7*b7)        ...        hi(a0*b0) |
    pmullw xmm0, xmm1     ; xmm0 = | low(a7*b7)       ...       low(a0*b0) |
```

Multiplicar vectores

```
mulvec: ; rdi = int16_t *v1, rsi = int16_t *v2, rdx = int32_t *resultado
push rbp
mov rbp,rsp
mov rcx, (128 >> 2) ; rcx = 128 / 8
.ciclo:
    movdqa xmm0, [rdi]    ; xmm0 = | a7 | a6 | a5 | a4 | a3 | a2 | a1 | a0 |
    movdqa xmm1, [rsi]    ; xmm1 = | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
    movdqa xmm2, xmm0     ; xmm2 = | a7 | a6 | a5 | a4 | a3 | a2 | a1 | a0 |
    pmulhw xmm2, xmm1     ; xmm2 = | hi(a7*b7)        ...        hi(a0*b0) |
    pmullw xmm0, xmm1     ; xmm0 = | low(a7*b7)       ...       low(a0*b0) |
    movdqa xmm1, xmm0     ; xmm1 = | low(a7*b7)       ...       low(a0*b0) |
    punpcklwd xmm0, xmm2  ; xmm0 = | hi:low(a3*b3)    ...    hi:low(0a*b0) |
    punpckhwd xmm1, xmm2  ; xmm1 = | hi:low(a7*b7)    ...    hi:low(a4*b4) |
```

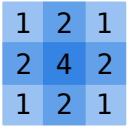
Multiplicar vectores

```
mulvec: ; rdi = int16_t *v1, rsi = int16_t *v2, rdx = int32_t *resultado
push rbp
mov rbp,rsp
mov rcx, (128 >> 2) ; rcx = 128 / 8
.ciclo:
    movdqa xmm0, [rdi]    ; xmm0 = | a7 | a6 | a5 | a4 | a3 | a2 | a1 | a0 |
    movdqa xmm1, [rsi]    ; xmm1 = | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
    movdqa xmm2, xmm0     ; xmm2 = | a7 | a6 | a5 | a4 | a3 | a2 | a1 | a0 |
    pmulhw xmm2, xmm1     ; xmm2 = | hi(a7*b7)        ...        hi(a0*b0) |
    pmullw xmm0, xmm1     ; xmm0 = | low(a7*b7)       ...       low(a0*b0) |
    movdqa xmm1, xmm0     ; xmm1 = | low(a7*b7)       ...       low(a0*b0) |
    punpcklwd xmm0, xmm2  ; xmm0 = | hi:low(a3*b3)    ...    hi:low(0a*b0) |
    punpckhwd xmm1, xmm2  ; xmm1 = | hi:low(a7*b7)    ...    hi:low(a4*b4) |
    movdqa [rdx], xmm0
    movdqa [rdx+16], xmm1
```

Multiplicar vectores

```
mulvec: ; rdi = int16_t *v1, rsi = int16_t *v2, rdx = int32_t *resultado
push rbp
mov rbp,rsp
mov rcx, (128 >> 2) ; rcx = 128 / 8
.ciclo:
    movdqa xmm0, [rdi]    ; xmm0 = | a7 | a6 | a5 | a4 | a3 | a2 | a1 | a0 |
    movdqa xmm1, [rsi]    ; xmm1 = | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
    movdqa xmm2, xmm0     ; xmm2 = | a7 | a6 | a5 | a4 | a3 | a2 | a1 | a0 |
    pmulhw xmm2, xmm1     ; xmm2 = | hi(a7*b7)        ...        hi(a0*b0) |
    pmullw xmm0, xmm1     ; xmm0 = | low(a7*b7)       ...       low(a0*b0) |
    movdqa xmm1, xmm0     ; xmm1 = | low(a7*b7)       ...       low(a0*b0) |
    punpcklwd xmm0, xmm2  ; xmm0 = | hi:low(a3*b3)    ...    hi:low(0a*b0) |
    punpckhwd xmm1, xmm2  ; xmm1 = | hi:low(a7*b7)    ...    hi:low(a4*b4) |
    movdqa [rdx], xmm0
    movdqa [rdx+16], xmm1
    add rdx, 32
    add rdi, 16
    add rsi, 16
loop .ciclo
pop rbp
ret
```

Técnica: Operatoria con un kernel

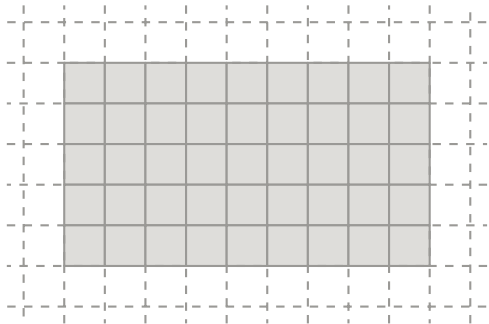


Técnica: Operatoria con un kernel

$$\left(\texttt{datos} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \right) / 16$$

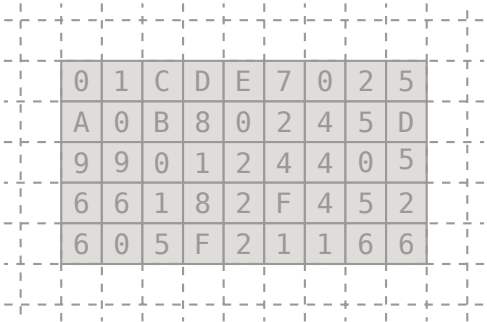
Técnica: Operatoria con un kernel

$$\left(\text{datos} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \right) / 16$$



Técnica: Operatoria con un kernel

$$\left(\text{datos} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \right) / 16$$



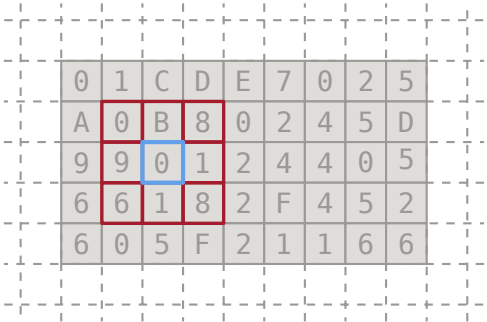
Técnica: Operatoria con un kernel

$$\left(\text{datos} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \right) / 16$$

0	1	C	D	E	7	0	2	5
A	0	B	8	0	2	4	5	D
9	9	0	1	2	4	4	0	5
6	6	1	8	2	F	4	5	2
6	0	5	F	2	1	1	6	6

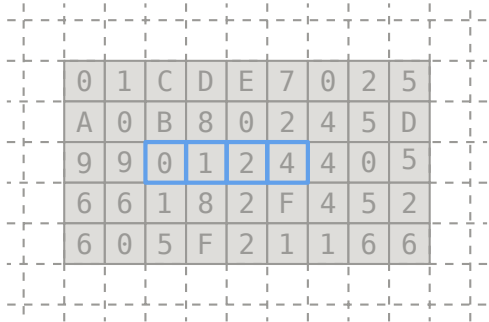
Técnica: Operatoria con un kernel

$$\left(\text{datos} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \right) / 16$$



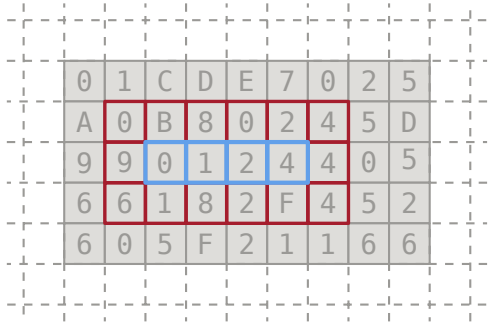
Técnica: Operatoria con un kernel

$$\left(\text{datos} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \right) / 16$$



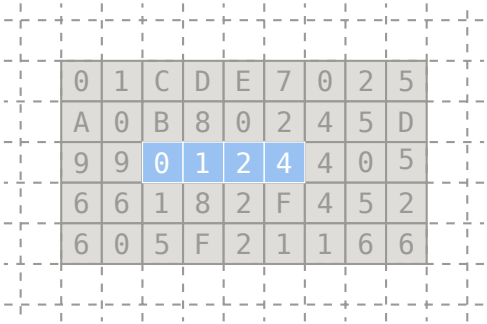
Técnica: Operatoria con un kernel

$$\left(\text{datos} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \right) / 16$$

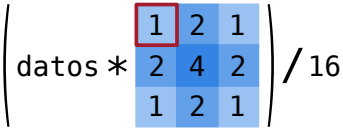


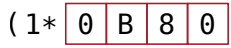
Técnica: Operatoria con un kernel

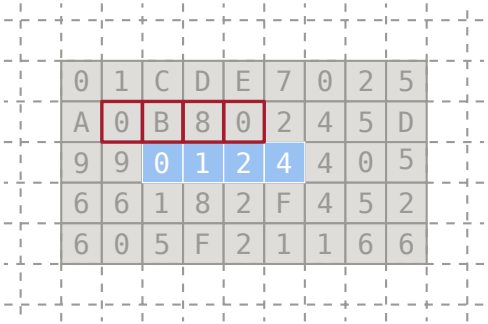
$$\left(\text{datos} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \right) / 16$$



Técnica: Operatoria con un kernel



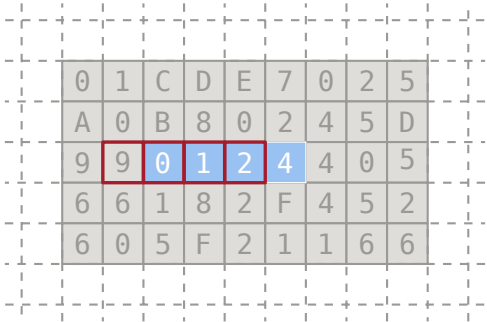




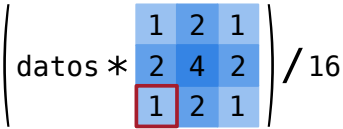
Técnica: Operatoria con un kernel

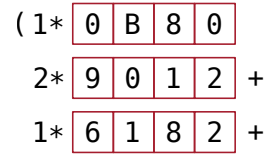
$$\left(\text{datos} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \right) / 16$$

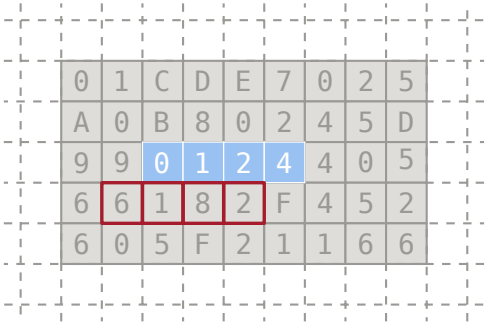
$$(1 * \begin{bmatrix} 0 & B & 8 & 0 \end{bmatrix} + 2 * \begin{bmatrix} 9 & 0 & 1 & 2 \end{bmatrix} +$$



Técnica: Operatoria con un kernel







Técnica: Operatoria con un kernel

$$\left(\text{datos} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \right) / 16$$

	0	1	C	D	E	7	0	2	5
	A	0	B	8	0	2	4	5	D
	9	9	0	1	2	4	4	0	5
	6	6	1	8	2	F	4	5	2
	6	0	5	F	2	1	1	6	6

(1* 0 B 8 0
2* 9 0 1 2 +
1* 6 1 8 2 +
2* B 8 0 2 +

Técnica: Operatoria con un kernel

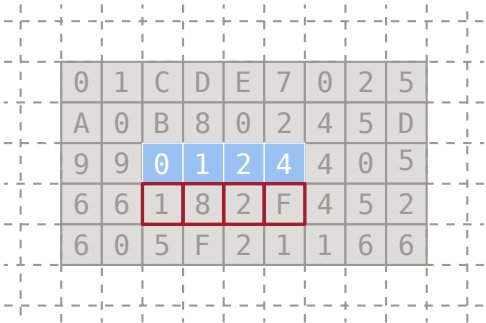
$$\left(\text{datos} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \right) / 16$$

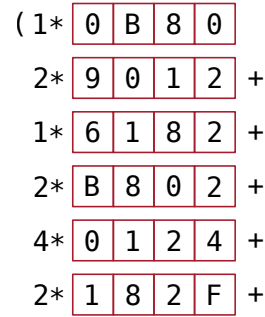
	0	1	C	D	E	7	0	2	5
	A	0	B	8	0	2	4	5	D
	9	9	0	1	2	4	4	0	5
	6	6	1	8	2	F	4	5	2
	6	0	5	F	2	1	1	6	6

$$\begin{aligned} (1 * \begin{bmatrix} 0 & B & 8 & 0 \end{bmatrix} \\ 2 * \begin{bmatrix} 9 & 0 & 1 & 2 \end{bmatrix} + \\ 1 * \begin{bmatrix} 6 & 1 & 8 & 2 \end{bmatrix} + \\ 2 * \begin{bmatrix} B & 8 & 0 & 2 \end{bmatrix} + \\ 4 * \begin{bmatrix} 0 & 1 & 2 & 4 \end{bmatrix} + \end{aligned}$$

Técnica: Operatoria con un kernel

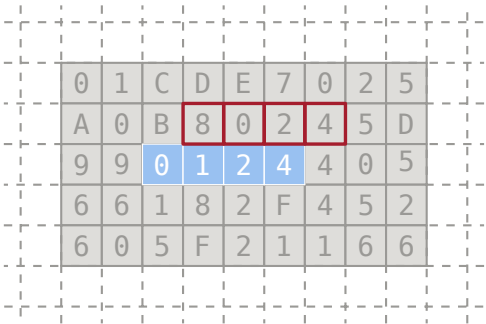
$$\left(\text{datos} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \right) / 16$$





Técnica: Operatoria con un kernel

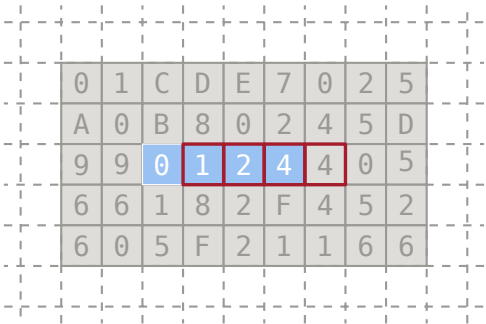
$$\left(\text{datos} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \right) / 16$$

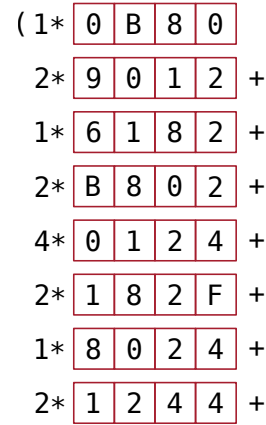


$$\begin{aligned}
(1*&\begin{bmatrix} 0 & B & 8 & 0 \end{bmatrix} \\
2*&\begin{bmatrix} 9 & 0 & 1 & 2 \end{bmatrix} + \\
1*&\begin{bmatrix} 6 & 1 & 8 & 2 \end{bmatrix} + \\
2*&\begin{bmatrix} B & 8 & 0 & 2 \end{bmatrix} + \\
4*&\begin{bmatrix} 0 & 1 & 2 & 4 \end{bmatrix} + \\
2*&\begin{bmatrix} 1 & 8 & 2 & F \end{bmatrix} + \\
1*&\begin{bmatrix} 8 & 0 & 2 & 4 \end{bmatrix} +
\end{aligned}$$

Técnica: Operatoria con un kernel

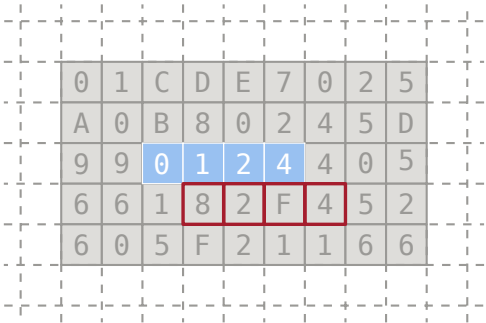
$$\left(\text{datos} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \right) / 16$$





Técnica: Operatoria con un kernel

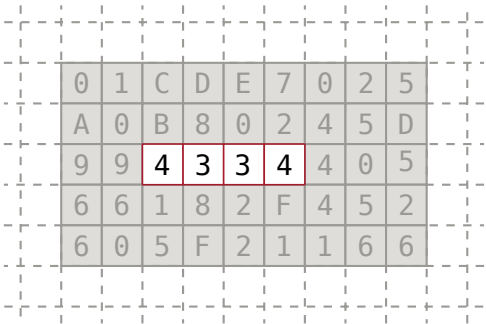
$$\left(\text{datos} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \right) / 16$$



$$
\begin{aligned}
(1*&\begin{bmatrix} 0 & B & 8 & 0 \end{bmatrix} \\
2*&\begin{bmatrix} 9 & 0 & 1 & 2 \end{bmatrix} + \\
1*&\begin{bmatrix} 6 & 1 & 8 & 2 \end{bmatrix} + \\
2*&\begin{bmatrix} B & 8 & 0 & 2 \end{bmatrix} + \\
4*&\begin{bmatrix} 0 & 1 & 2 & 4 \end{bmatrix} + \\
2*&\begin{bmatrix} 1 & 8 & 2 & F \end{bmatrix} + \\
1*&\begin{bmatrix} 8 & 0 & 2 & 4 \end{bmatrix} + \\
2*&\begin{bmatrix} 1 & 2 & 4 & 4 \end{bmatrix} + \\
1*&\begin{bmatrix} 8 & 2 & F & 4 \end{bmatrix})
\end{aligned}
$$

Técnica: Operatoria con un kernel

$$\left(\text{datos} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \right) / 16$$



$$
\begin{aligned}
(1 * & \begin{bmatrix} 0 & B & 8 & 0 \end{bmatrix} \\
2 * & \begin{bmatrix} 9 & 0 & 1 & 2 \end{bmatrix} + \\
1 * & \begin{bmatrix} 6 & 1 & 8 & 2 \end{bmatrix} + \\
2 * & \begin{bmatrix} B & 8 & 0 & 2 \end{bmatrix} + \\
4 * & \begin{bmatrix} 0 & 1 & 2 & 4 \end{bmatrix} + \\
2 * & \begin{bmatrix} 1 & 8 & 2 & F \end{bmatrix} + \\
1 * & \begin{bmatrix} 8 & 0 & 2 & 4 \end{bmatrix} + \\
2 * & \begin{bmatrix} 1 & 2 & 4 & 4 \end{bmatrix} + \\
1 * & \begin{bmatrix} 8 & 2 & F & 4 \end{bmatrix}) \\
/16 = & \begin{bmatrix} 4 & 3 & 3 & 4 \end{bmatrix}
\end{aligned}
$$

Ejemplo

Efecto Blur

Aplicar un kernel de 3×3 `[[1,2,1],[2,4,2],[1,2,1]] / 16` sobre cada pixel de una imagen de 34×34 de valores de 8 bits. Almacenar el resultado en una imagen de 32×32.

```
extern void blur(uint8_t *imgDst, uint8_t *imgSrc)
```

Efecto Blur (Parte 1/5)

```
; rdi = uint8_t *imgDst
; rsi = uint8_t *imgSrc

blur:
  push rbp
  mov rbp,rsp
```

Efecto Blur (Parte 1/5)

```
; rdi = uint8_t *imgDst
; rsi = uint8_t *imgSrc

blur:
  push rbp
  mov rbp,rsp

  mov rdx, 32          ; rdx = 32 (filas)
  lea rsi, [rsi+34+1]  ; rsi = primera fila mas uno
```

Efecto Blur (Parte 1/5)

```
; rdi = uint8_t *imgDst
; rsi = uint8_t *imgSrc

blur:
  push rbp
  mov rbp,rsp

  mov rdx, 32          ; rdx = 32 (filas)
  lea rsi, [rsi+34+1]  ; rsi = primera fila mas uno

  .cicloFilas:
    mov rcx, 32 >> 3   ; rcx = 32 / 8 (columnas)
```

Efecto Blur (Parte 1/5)

```
; rdi = uint8_t *imgDst
; rsi = uint8_t *imgSrc

blur:
  push rbp
  mov rbp,rsp

  mov rdx, 32          ; rdx = 32 (filas)
  lea rsi, [rsi+34+1]  ; rsi = primera fila mas uno

  .cicloFilas:
    mov rcx, 32 >> 3   ; rcx = 32 / 8 (columnas)

      .cicloColumnas:
        pxor xmm0, xmm0 ; xmm0 = 0 (acumulador)
```

Efecto Blur (Parte 1/5)

```
; rdi = uint8_t *imgDst
; rsi = uint8_t *imgSrc

blur:
  push rbp
  mov rbp,rsp

  mov rdx, 32          ; rdx = 32 (filas)
  lea rsi, [rsi+34+1]  ; rsi = primera fila mas uno

  .cicloFilas:
    mov rcx, 32 >> 3   ; rcx = 32 / 8 (columnas)

      .cicloColumnas:
        pxor xmm0, xmm0 ; xmm0 = 0 (acumulador)

        ; 1*A 2*D 1*G
        ; 2*B 4*E 2*H
        ; 1*C 2*F 1*I
```

Efecto Blur (Parte 2/5)

```
; 1*A 2*D 1*G
; 2*B 4*E 2*H
; 1*C 2*F 1*I
```

Efecto Blur (Parte 2/5)

```
; 1*A 2*D 1*G
; 2*B 4*E 2*H
; 1*C 2*F 1*I

pmovzxbw xmm1, [rsi-1-34] ; xmm1 = A
pmovzxbw xmm2, [rsi-1]    ; xmm2 = B
pmovzxbw xmm3, [rsi-1+34] ; xmm3 = C
```

Efecto Blur (Parte 2/5)

```
; 1*A 2*D 1*G
; 2*B 4*E 2*H
; 1*C 2*F 1*I

pmovzxbw xmm1, [rsi-1-34] ; xmm1 = A
pmovzxbw xmm2, [rsi-1]    ; xmm2 = B
pmovzxbw xmm3, [rsi-1+34] ; xmm3 = C

psllw xmm2, 1             ; xmm2 = 2*B
```

Efecto Blur (Parte 2/5)

```
; 1*A 2*D 1*G
; 2*B 4*E 2*H
; 1*C 2*F 1*I

pmovzxbw xmm1, [rsi-1-34] ; xmm1 = A
pmovzxbw xmm2, [rsi-1]    ; xmm2 = B
pmovzxbw xmm3, [rsi-1+34] ; xmm3 = C

psllw xmm2, 1             ; xmm2 = 2*B

paddw xmm0, xmm1
paddw xmm0, xmm2
paddw xmm0, xmm3
```

Efecto Blur (Parte 3/5)

```
; 1*A 2*D 1*G
; 2*B 4*E 2*H
; 1*C 2*F 1*I
```

Efecto Blur (Parte 3/5)

```
; 1*A 2*D 1*G
; 2*B 4*E 2*H
; 1*C 2*F 1*I

pmovzxbw xmm1, [rsi-34]   ; xmm1 = D
pmovzxbw xmm2, [rsi]      ; xmm2 = E
pmovzxbw xmm3, [rsi+34]   ; xmm3 = F
```

Efecto Blur (Parte 3/5)

```
; 1*A 2*D 1*G
; 2*B 4*E 2*H
; 1*C 2*F 1*I

pmovzxbw xmm1, [rsi-34]   ; xmm1 = D
pmovzxbw xmm2, [rsi]      ; xmm2 = E
pmovzxbw xmm3, [rsi+34]   ; xmm3 = F

psllw xmm1, 1             ; xmm1 = 2*D
psllw xmm2, 2             ; xmm2 = 4*E
psllw xmm3, 1             ; xmm3 = 2*F
```

Efecto Blur (Parte 3/5)

```
; 1*A 2*D 1*G
; 2*B 4*E 2*H
; 1*C 2*F 1*I

pmovzxbw xmm1, [rsi-34]   ; xmm1 = D
pmovzxbw xmm2, [rsi]      ; xmm2 = E
pmovzxbw xmm3, [rsi+34]   ; xmm3 = F

psllw xmm1, 1             ; xmm1 = 2*D
psllw xmm2, 2             ; xmm2 = 4*E
psllw xmm3, 1             ; xmm3 = 2*F

paddw xmm0, xmm1
paddw xmm0, xmm2
paddw xmm0, xmm3
```

Efecto Blur (Parte 4/5)

```
; 1*A 2*D 1*G
; 2*B 4*E 2*H
; 1*C 2*F 1*I
```

Efecto Blur (Parte 4/5)

```
; 1*A 2*D 1*G
; 2*B 4*E 2*H
; 1*C 2*F 1*I

pmovzxbw xmm1, [rsi+1-34] ; xmm1 = G
pmovzxbw xmm2, [rsi+1]    ; xmm2 = H
pmovzxbw xmm3, [rsi+1+34] ; xmm3 = I
```

Efecto Blur (Parte 4/5)

```
; 1*A 2*D 1*G
; 2*B 4*E 2*H
; 1*C 2*F 1*I

pmovzxbw xmm1, [rsi+1-34] ; xmm1 = G
pmovzxbw xmm2, [rsi+1]    ; xmm2 = H
pmovzxbw xmm3, [rsi+1+34] ; xmm3 = I

psllw xmm2, 1             ; xmm2 = 2*H
```

Efecto Blur (Parte 4/5)

```
; 1*A 2*D 1*G
; 2*B 4*E 2*H
; 1*C 2*F 1*I

pmovzxbw xmm1, [rsi+1-34] ; xmm1 = G
pmovzxbw xmm2, [rsi+1]    ; xmm2 = H
pmovzxbw xmm3, [rsi+1+34] ; xmm3 = I

psllw xmm2, 1             ; xmm2 = 2*H

paddw xmm0, xmm1
paddw xmm0, xmm2
paddw xmm0, xmm3
```

Efecto Blur (Parte 5/5)

```
psrlw xmm0, 4
packuswb xmm0, xmm0
```

Efecto Blur (Parte 5/5)

```
psrlw xmm0, 4
packuswb xmm0, xmm0

movq [rdi], xmm0
add rdi, 8
add rsi, 8
```

Efecto Blur (Parte 5/5)

```
    psrlw xmm0, 4
    packuswb xmm0, xmm0

    movq [rdi], xmm0
    add rdi, 8
    add rsi, 8

  dec rcx
  cmp rcx, 0
  jnz .cicloColumnas
  lea rsi, [rsi+2]
```

Efecto Blur (Parte 5/5)

```
        psrlw xmm0, 4
        packuswb xmm0, xmm0

        movq [rdi], xmm0
        add rdi, 8
        add rsi, 8

      dec rcx
      cmp rcx, 0
      jnz .cicloColumnas
      lea rsi, [rsi+2]

  dec rdx
  cmp rdx, 0
  jnz .cicloFilas

  pop rbp
ret
```

Efecto Blur

```
; rdi = uint8_t *imgDst
; rsi = uint8_t *imgSrc

blur:
  push rbp
  mov rbp,rsp

  pxor xmm8, xmm8
  mov rdx, 32
  lea rsi, [rsi+34+1]

  .cicloFilas:
    mov rcx, 32 >> 3

      .cicloColumnas:
        pxor xmm0, xmm0

        ; 1*A 2*D 1*G
        ; 2*B 4*E 2*H
        ; 1*C 2*F 1*I

        pmovzxbw xmm1, [rsi-1-34] ; xmm1 = A
        pmovzxbw xmm2, [rsi-1]    ; xmm2 = B
        pmovzxbw xmm3, [rsi-1+34] ; xmm3 = C
        psllw xmm2, 1             ; xmm2 = 2*B
        paddw xmm0, xmm1
        paddw xmm0, xmm2
        paddw xmm0, xmm3

        pmovzxbw xmm1, [rsi-34]   ; xmm1 = D
        pmovzxbw xmm2, [rsi]      ; xmm2 = E
        pmovzxbw xmm3, [rsi+34]   ; xmm3 = F
        psllw xmm1, 1             ; xmm1 = 2*D
        psllw xmm2, 2             ; xmm2 = 4*E
        psllw xmm3, 1             ; xmm3 = 2*F
        paddw xmm0, xmm1
        paddw xmm0, xmm2
        paddw xmm0, xmm3

        pmovzxbw xmm1, [rsi+1-34] ; xmm1 = G
        pmovzxbw xmm2, [rsi+1]    ; xmm2 = H
        pmovzxbw xmm3, [rsi+1+34] ; xmm3 = I
        psllw xmm2, 1             ; xmm2 = 2*H
        paddw xmm0, xmm1
        paddw xmm0, xmm2
        paddw xmm0, xmm3

        psrlw xmm0, 4
        packuswb xmm0, xmm0

        movq [rdi], xmm0
        add rdi, 8
        add rsi, 8

      dec rcx
      cmp rcx, 0
      jnz .cicloColumnas
      lea rsi, [rsi+2]

  dec rdx
  cmp rdx, 0
  jnz .cicloFilas

  pop rbp
ret
```

Ejercicios

1. Sean un vector de 1024 pares de componentes x e y, ordenadas una a continuación de la otra. Las componentes están almacenadas en punto flotante de 32 bits. Calcular el módulo del vector que representan utilizando la formula $\sqrt{x^2 + y^2}$. Retornar el resultado un nuevo vector. La aridad de la función es `float* mod(float *v)`.

2. Dados dos vectores de 64 enteros de 16 bits, realizar el producto escalar entre ambos y alamacenar el resultado en 32 bits. La aridad de la función es `int32_t dotProduct(int16_t *a, int16_t *b)`.

3. Dadas dos matrices de 32×32 valores en punto flotante de 32 bits, realizar la multiplicación de matrices entre ambas y almacenar el resultado en una nueva matriz solicitando memoria. La aridad de la función es `float* matrixProduct(float *a, float *b)`.

¡Gracias!

Recuerden leer los comentarios al final de
este video por aclaraciones o fe de erratas.