SIMD

Gonzalo Raposo

Organización del Computador II, 2do cuat. de 2014

Motivación: Procesamiento de imágenes

Incrementar el brillo de una imagen

Motivación: Procesamiento de imágenes

- Para hacer esto hay que *sumar* un valor fijo **b** a cada *pixel* de la imagen.

Incrementar brillo de una imagen

```
for(int f = 0; f < alto; f++)
  for(int c = 0; c < ancho; c++)
    I[f][c] += b
```

Procesamiento de imágenes

- Para una *sola* imagen, el proceso es *rápido*.
- Si tenemos un *video* y tenemos que realizar el proceso para cada uno de los frames, esto puede llegar a ser *lento*
- Para cada frame (imagen) estamos haciendo:

Incrementar brillo de un frame en un valor fijo b

$$
\begin{aligned}
frame_i[0][0] &\mathrel{+}= b \\
frame_i[0][1] &\mathrel{+}= b \\
&\dots \\
frame_i[alto-1][ancho-1] &\mathrel{+}= b
\end{aligned}
$$

Procesamiento de imágenes

- Estamos aplicando la misma operación a una *gran* cantidad de datos.
- Cómo podríamos acelerar este proceso?
- Y si sumamos de a más de un elemento a la vez?

$$\begin{pmatrix} I[0][0] \\ I[0][1] \\ I[0][2] \\ I[0][3] \end{pmatrix} + \begin{pmatrix} b \\ b \\ b \\ b \end{pmatrix} = \begin{pmatrix} I[0][0] + b \\ I[0][1] + b \\ I[0][2] + b \\ I[0][3] + b \end{pmatrix}$$

- En vez de sumar de a un elemento lo hacemos de a *bloques de 4* elementos.
- Obtenemos el mismo resultado, pero tenemos un rendimiento *4 veces superior*.

Procesamiento vectorial

- Esta manera de trabajar es lo que se conoce como **SIMD**:

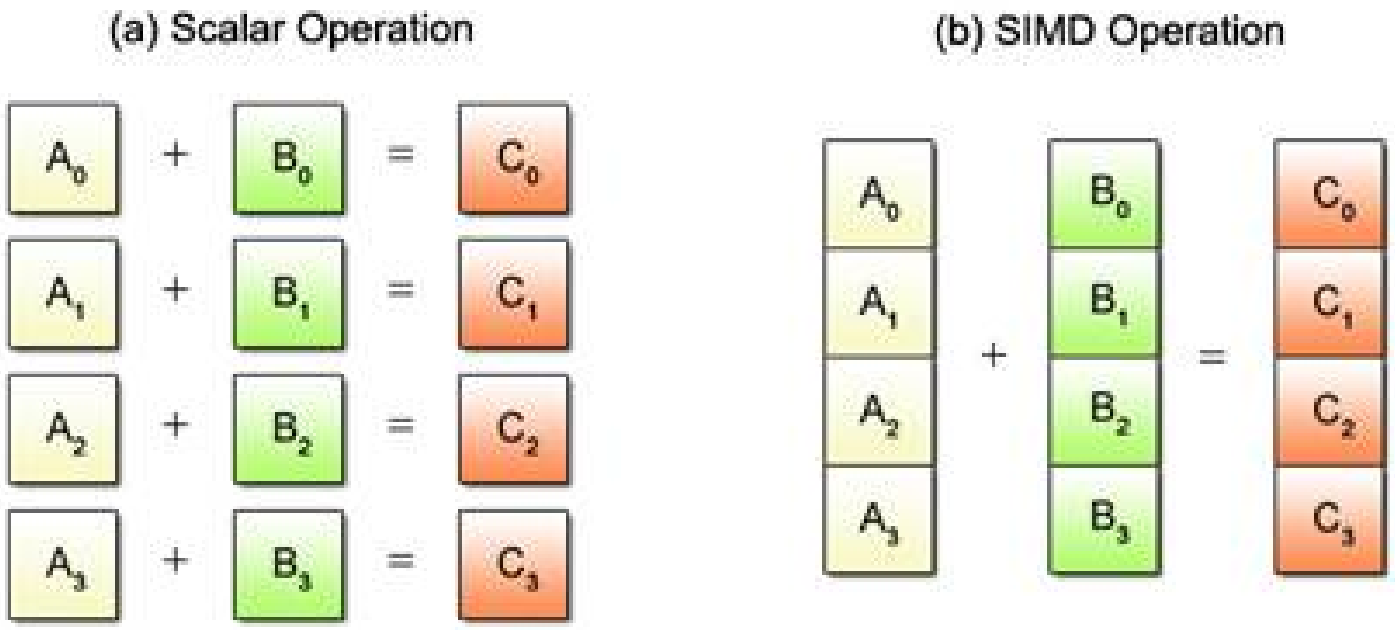
  Single **I**nstruction, **M**ultiple **D**ata.

- Hasta este momento veniamos ulitizando **SISD**:

  Single **I**nstrucction, **S**ingle **D**ata.

Procesamiento vectorial

Esquemáticamente:



Procesamiento vectorial

- Siempre podemos usar **SIMD**?

  No. Hay muchos algoritmos que no se pueden adaptar a este tipo de procesamiento. Por ejemplo, lo que hicieron en el **tp1**.

- En qué casos es útil?

  Para *procesamiento multimedia*, es decir, procesamiento de **imágenes**, **videos** y **audio**, y cualquier otro tipo de procesamiento que involucre aplicar la misma operación a una gran cantidad de datos.

Implementación del modelo SIMD

- **SSE** (**S**treaming **S**IMD **E**xtensions) es un set de instrucciones que implementa el modelo de cómputo **SIMD**.
- Se introdujo por primera vez en el año 1999 por **Intel** como sucesor de **MMX** (introducido en 1997).
- **SSE** extiende a **MMX** con nuevos registros y nuevos tipos de datos.
- En los últimos años se fue extendiendo el set de instrucciones de **SSE** hasta llegar a la versión **SSE4.2**

Tipos de datos / Registros

- **SSE** maneja los siguientes tipos de datos:
  1. **enteros**,
  2. **floats** y
  3. **doubles** (a partir de **SSE2**)

- y cuenta con 15 registros: **XMM0**, **XMM1**, ... , **XMM15**
  1. Son de **128 bits** (**16** bytes)
  2. Pueden almacenar los tipos de datos mencionados

Empaquetamiento

- **SSE** maneja los registro de manera diferente a lo que estamos acostumbrados.
- Por ejemplo, al hacer:

```
mov rax, 10
```

rax contiene *un solo dato* al cual le podemos aplicar distintas operaciones (sumar, restar, etc).

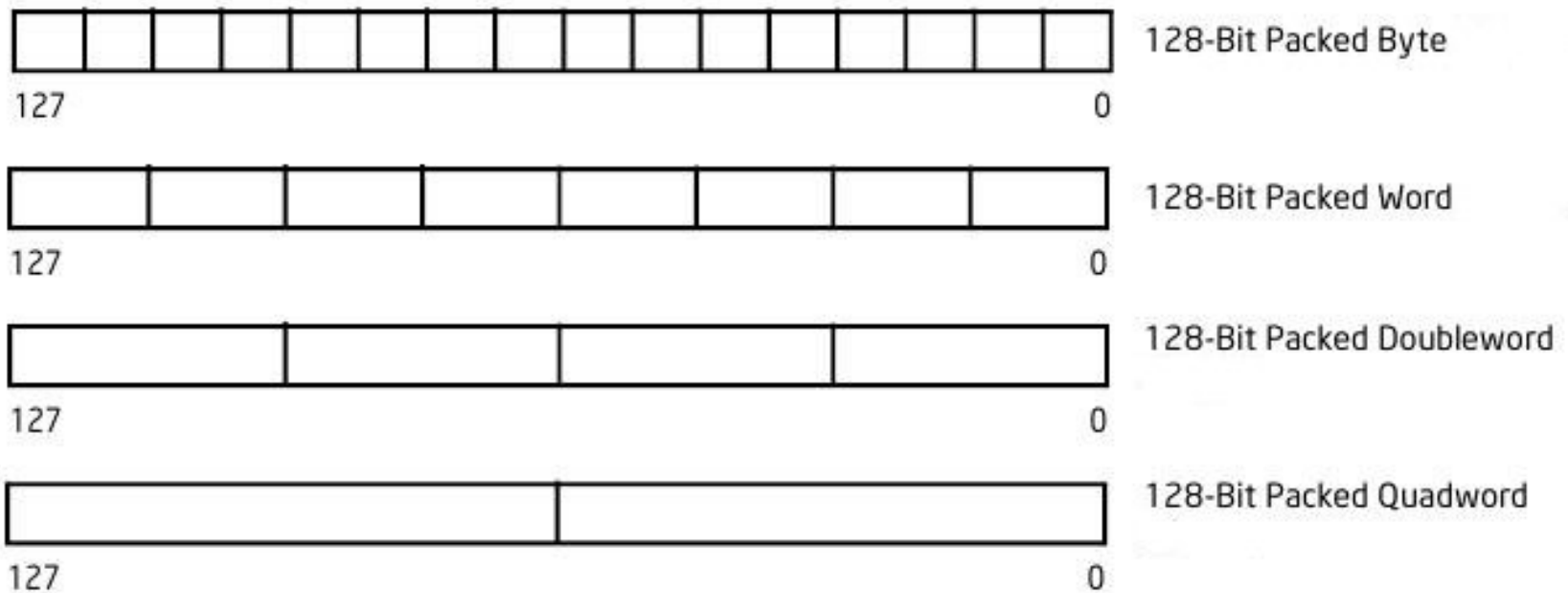
- En **SSE** esto es distinto. Tenemos registros en el cual ponemos *más de un dato* (siempre del *mismo tipo*).

Empaquetamiento

- En un registro de **SSE** (**16 bytes**) podemos poner:
  - ***16*** datos de tamaño **1 byte** (entero (**char**))
  - ***8*** datos de tamaño **2 bytes** (entero (**short**))
  - ***4*** datos de tamaño **4 bytes** (entero (**int**) o de punto flotante (**float**))
  - ***2*** datos de tamaño **8 bytes** (entero (**long long int**) o de punto flotante (**double**))
- Esto se lo conoce como *empaquetamiento*, más de un dato en un mismo registro.

Empaquetamiento

Esquemáticamente:



Empaquetamiento

- Cómo sabemos que tipo de datos tenemos dentro de un registro?

  No hay manera desde las instrucciones. Ustedes escriben el código, por ende, ustedes saben que están poniendo en cada registro.

¿Cómo operar sobre esos registros?

- El set de instrucciones **SSE** brinda muchas operaciones para trabajar con estos registros:
  - Movimiento de datos,
  - Aritméticas,
  - Lógicas,
  - de Comparación, etc.
- Hay instrucciones:
  - Para cada *tipo* de dato, y
  - cada *tamaño* de dato.

Instrucciones: ¿Cuál usar?

- La gran mayoría siguen una regla.
- Para enteros: empiezan con *P*, le sigue el nombre de la *operación* y terminan con el *tamaño* del dato
  - *PADDB*: suma de a **B**yte
  - *PADDW*: suma de a **W**ord
  - *PADDD*: suma de a **D**oubleword
  - *PADDQ*: suma de a **Q**uadword
- Y para punto flotante tenemos: nombre de la *operación*, *modo* de operación y *tamaño* del dato.
  - *ADDPS*: suma de a **F**loat
  - *ADDPD*: suma de a **D**ouble

  La **P** proviene de **P**acked.

Instrucciones: ¿Cómo se usan?

- Continuando con el ejemplo del comienzo de la clase, supongamos que en **XMM0** tenemos los 16 primeros pixeles de la imagen y en **XMM1** tenemos **b** repetido 16 veces, entonces hacer:

paddb xmm1, xmm2

es equivalente a hacer:

$$
\begin{aligned}
I[0][0] &\mathrel{+}= b \\
I[0][1] &\mathrel{+}= b \\
&\dots \\
I[0][15] &\mathrel{+}= b
\end{aligned}
$$

Es decir, incrementamos la performance en **16x**!!! (en teoría, por lo menos...)

Ejercicio 1

Realizar el producto interno de dos vectores de floats.

- La longitud de ambos vectores es **n**, donde n es un short.
- n es múltiplo de 4.
- El prototipo de la función es:

```
float productoInternoFloats(float* a, float* b, short n)
```

Producto Interno

$$\langle a, b \rangle = \sum_{i=0}^{n-1} a[\,i\,] * b[\,i\,]$$

Ejercicio 1: Solución

```
productoInternoFloats:      ; rdi = a, rsi = b; dx = n

xor rcx, rcx                ; Contador
mov cx, dx
shr rcx, 2                  ; Proceso de a 4 elementos

pxor xmm7, xmm7             ; Acumulador

.ciclo:
                            ; Cargar los valores
    movups xmm1, [rdi]      ; xmm1 = a3 | a2 | a1 | a0
    movups xmm2, [rsi]      ; xmm2 = b3 | b2 | b1 | b0

                            ; Multiplicar
    mulps xmm1, xmm2        ; xmm1 = a3 * b3 | a2 * b2 | a1 * b1 | a0 * b0

                            ; Acumular el resultado
    addps xmm7, xmm1        ; xmm7 = sum3 | sum2 | sum1 | sum0
```

Ejercicio 1: Solución

```
    add rdi, 16             ; Avanzar los punteros
    add rsi, 16

    loop .ciclo
                            ; Sumar todo
movups xmm6, xmm7           ; xmm6 = sum3 | sum2 | sum1 | sum0
psrldq xmm6, 8              ; xmm6 = 0 | 0 | sum3 | sum2
addps xmm7, xmm6            ; xmm7 = ... | ... | sum3 + sum1 | sum2 + sum0

movups xmm6, xmm7           ; xmm6 = ... | ... | sum3 + sum1 | sum2 + sum0
psrldq xmm6, 4              ; xmm6 = ... | ... | ... | sum3 + sum1
addps xmm7, xmm6            ; xmm7 = ... | ... | ... | sumatoria

pxor xmm0, xmm0             ; xmm0 = 0 | 0 | 0 | 0
movss xmm0, xmm7            ; xmm0 = 0 | 0 | 0 | sumatoria

ret
```

Ejercicio 2

Realizar el producto interno de dos vectores de shorts.

- La longitud de ambos vectores es **n**, donde n es un short.
- n es múltiplo de 8.
- El prototipo de la función es:

```
int productoInternoShorts(short* a, short* b, short n)
```

Ejercicio 2: Solución

```
productoInternoShorts:    ; rdi = a, rsi = b; dx = n

xor rcx, rcx              ; Contador
mov cx, dx
shr rcx, 3                ; Proceso de a 8 elementos

pxor xmm7, xmm7           ; Acumulador

ciclo:
                          ; Cargar los valores
    movdqu xmm1, [rdi]    ; xmm1 = a7 | a6 | ... | a1 | a0
    movdqu xmm2, [rsi]    ; xmm2 = b7 | b6 | ... | b1 | b0
```

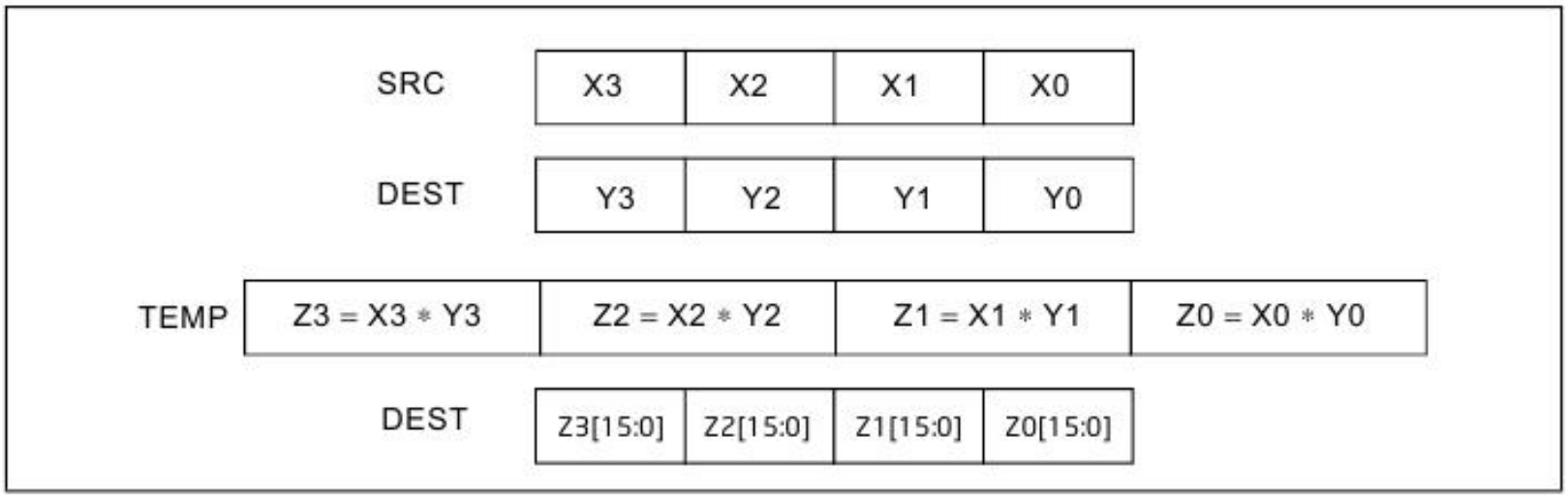
Ejercicio 2: ¿Cómo multiplicamos?

Para esto, tenemos las siguientes instrucciones de SSE:

- *pmullw* multiplica de a word y se queda con la *parte baja*.
- *pmulhw* multiplica de a word y se queda con la *parte alta*.

Ejercicio 2: ¿Cómo multiplicamos?

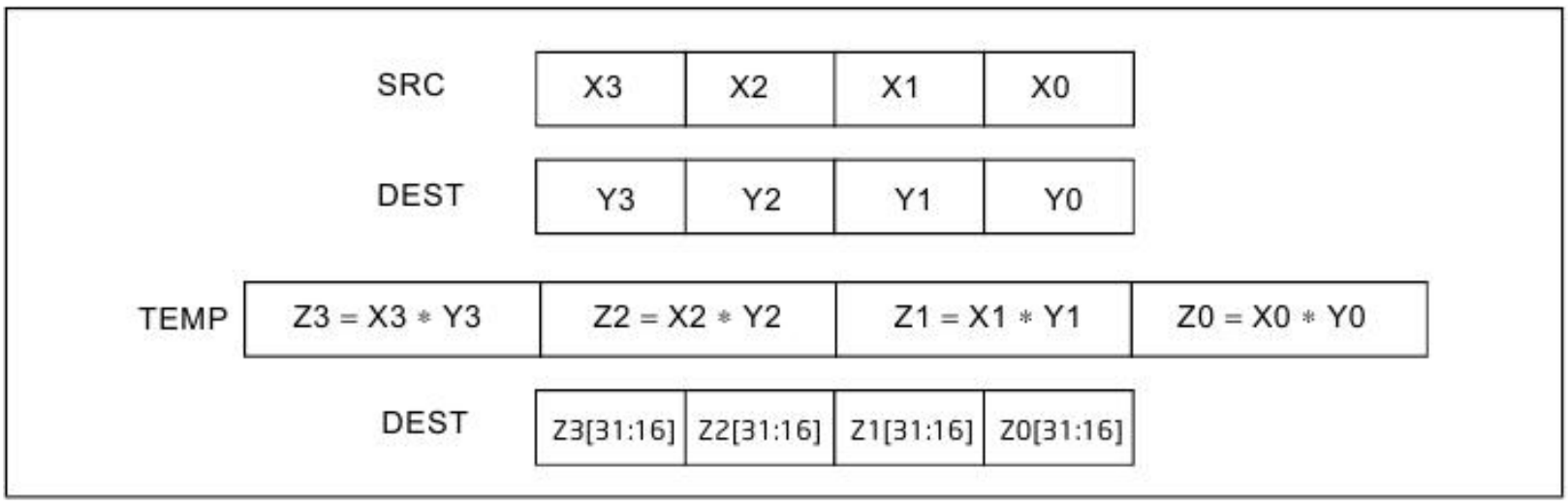
pmullw dst, src



(a) pmullw - Packed Multiply Low

Ejercicio 2: ¿Cómo multiplicamos?

pmulhw dst, src



(b) pmulhw - Packed Multiply High

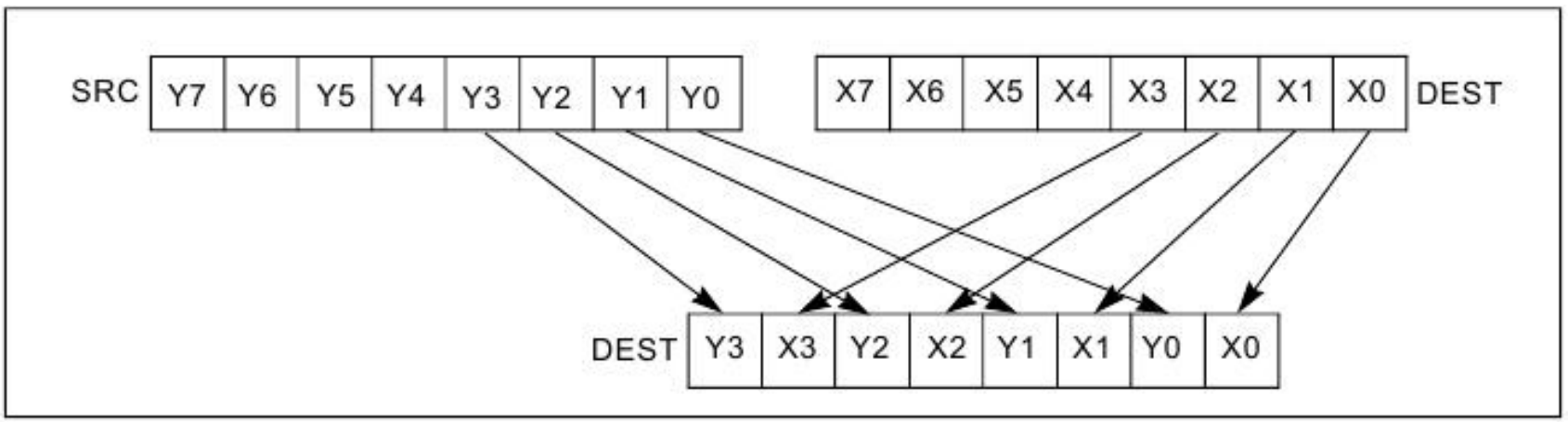
Ejercicio 2: ¿Cómo junto el resultado?

Para esto, tenemos las siguientes instrucciones de SSE:

- *punpcklwd* usa la *parte baja* de los registros para juntarlos.
- *punpckhwd* usa la *parte alta* de los registros para juntarlos.

Ejercicio 2: Desempaquetamiento

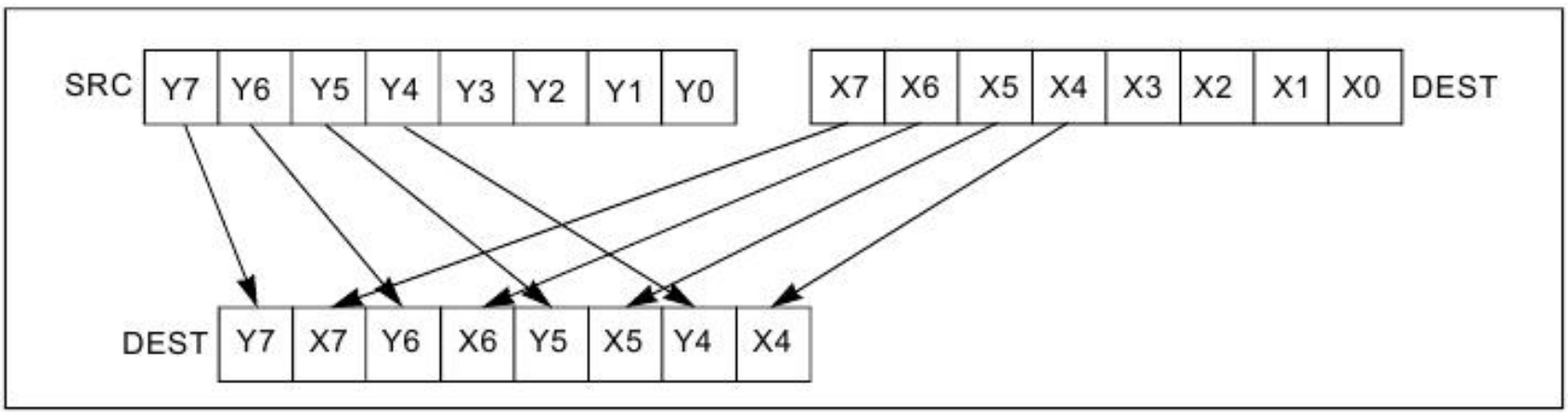
punpcklbw dst, src



(c) punpcklbw - Unpack Low Packed Data (Bytes to Words)

Ejercicio 2: Desempaquetamiento

punpckhbw dst, src



(d) punpckhbw - Unpack High Packed Data (Bytes to Words)

Ejercicio 2: Multiplicación

Multiplicamos:

```
movdqu xmm3, xmm1
pmullw xmm1, xmm2     ; xmm1 = parte baja de la multiplicación
pmulhw xmm3, xmm2     ; xmm3 = parte alta de la multiplicación
```

Juntamos el resultado:

```
movdqu xmm4, xmm1
punpcklwd xmm1, xmm3     ; xmm1 = a3 * b3 | a2 * b2 | a1 * b1 | a0 * b0
punpckhwd xmm4, xmm3     ; xmm4 = a7 * b7 | a6 * b6 | a5 * b5 | a4 * b4
```

Ejercicio 2: Solución.

```
productoInternoShorts:

xor rcx, rcx
mov cx, dx
shr rcx, 3

pxor xmm7, xmm7

.ciclo:
    movdqu xmm1, [rdi]
    movdqu xmm2, [rsi]

    movdqu xmm3, xmm1
    pmullw xmm1, xmm2
    pmulhw xmm3, xmm2

    movdqu xmm4, xmm1
    punpcklwd xmm1, xmm3
    punpckhwd xmm4, xmm3

    paddd xmm7, xmm1
    paddd xmm7, xmm4

    add rdi, 16
    add rsi, 16

    loop .ciclo

movdqu xmm6, xmm7
psrldq xmm6, 8
paddw xmm7, xmm6

movdqu xmm6, xmm7
psrldq xmm6, 4
paddw xmm7, xmm6

xor rax, rax
movd eax, xmm7

ret
```

Ejercicio 2: Una vuelta más...

- Y si uno de los vectores es de enteros de *1 byte*?
- ¿Cómo hacemos?

¿Preguntas?