Introducción a SIMD

Organización del Computador II

Gonza → Daniel N. Kundro → Facundo Ruiz

Departamento de Computación - FCEyN UBA

Segundo cuatrimestre de 2019

Hoy

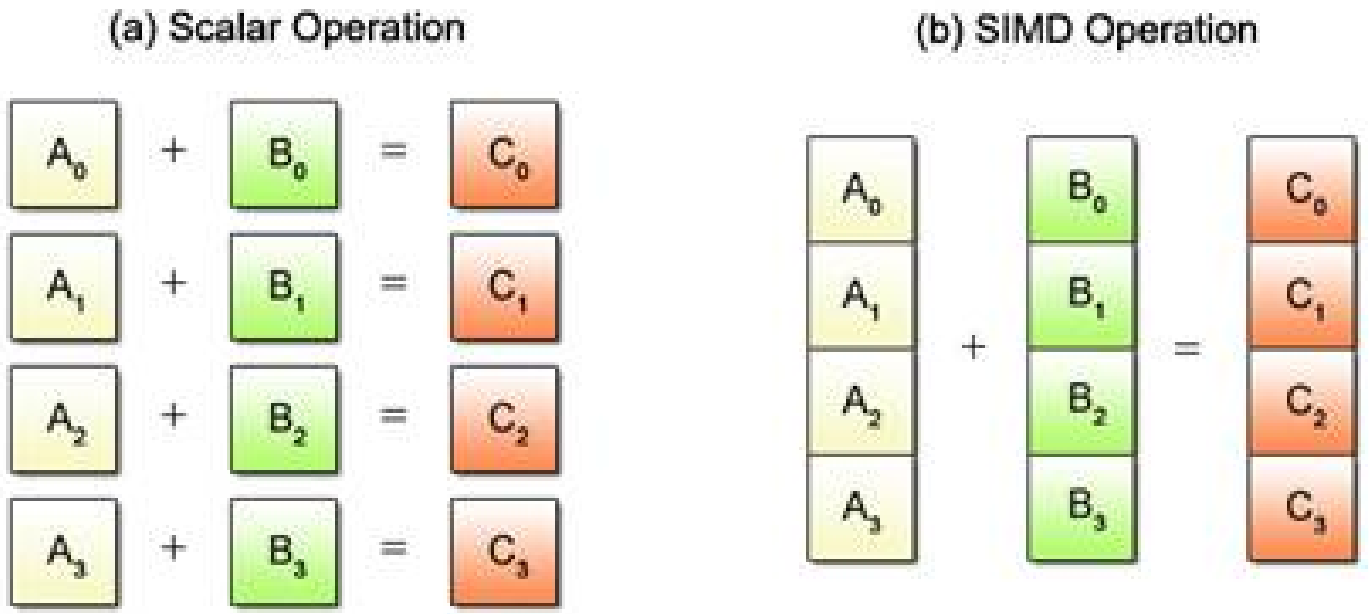
- Características de SIMD
- Tipos de datos y registros
- Algunas instrucciones
- Empaquetado/desempaquetado y comparación
- Cálculo de padding

Procesamiento vectorial, ¿qué vamos a ver?

Single **I**nstruction, **S**ingle **D**ata

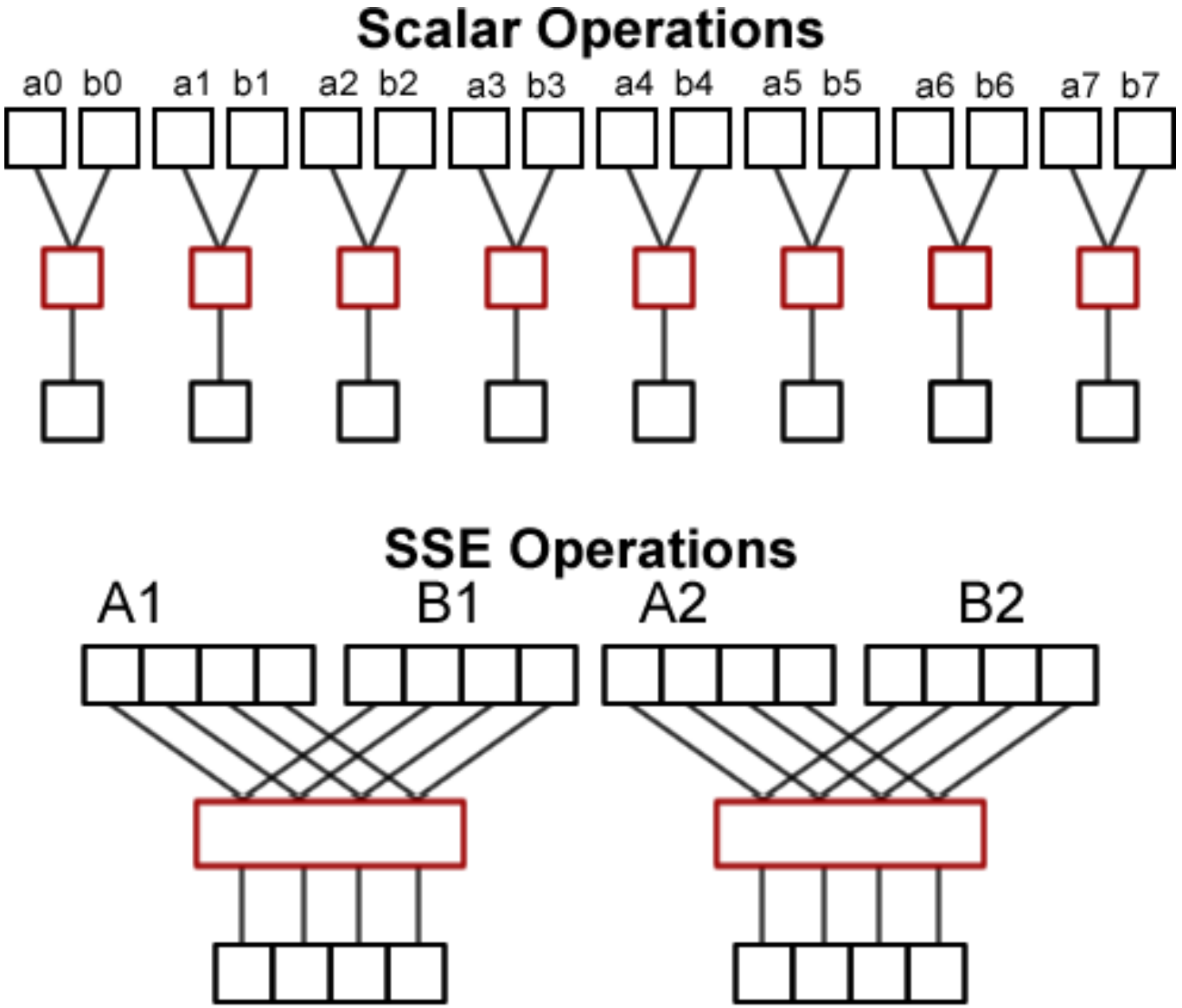
vs

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



El objetivo de SIMD es paralelizar operaciones a nivel de instrucción

Procesamiento vectorial, ¿qué vamos a ver?



Procesamiento vectorial, ¿para qué sirve?

- ¿Siempre podemos usar **SIMD**?

  NO. Hay muchos algoritmos que no se pueden adaptar a este tipo de procesamiento. Por ejemplo, lo que están haciendo 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 sobre una gran cantidad de datos

Implementación del modelo SIMD

- **SSE** (**S**treaming **SI**MD **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
- **SSE** se fue extendiendo hasta llegar a la versión **SSE4.2** con nuevas instrucciones
- Además en nuevos procesadores se implementó un nuevo set de instrucciones denominado **AVX**

Tipos de datos / Registros

SSE puede operar con los siguientes tipos de datos

- **enteros** (de 8, 16, 32, 64 y 128 bits)
- **floats**
- **doubles** (a partir de **SSE2**)

Cuenta con 16 registros de **128 bits** (**16** bytes)

- **XMM0**, **XMM1**, ..., **XMM15**

Empaquetamiento

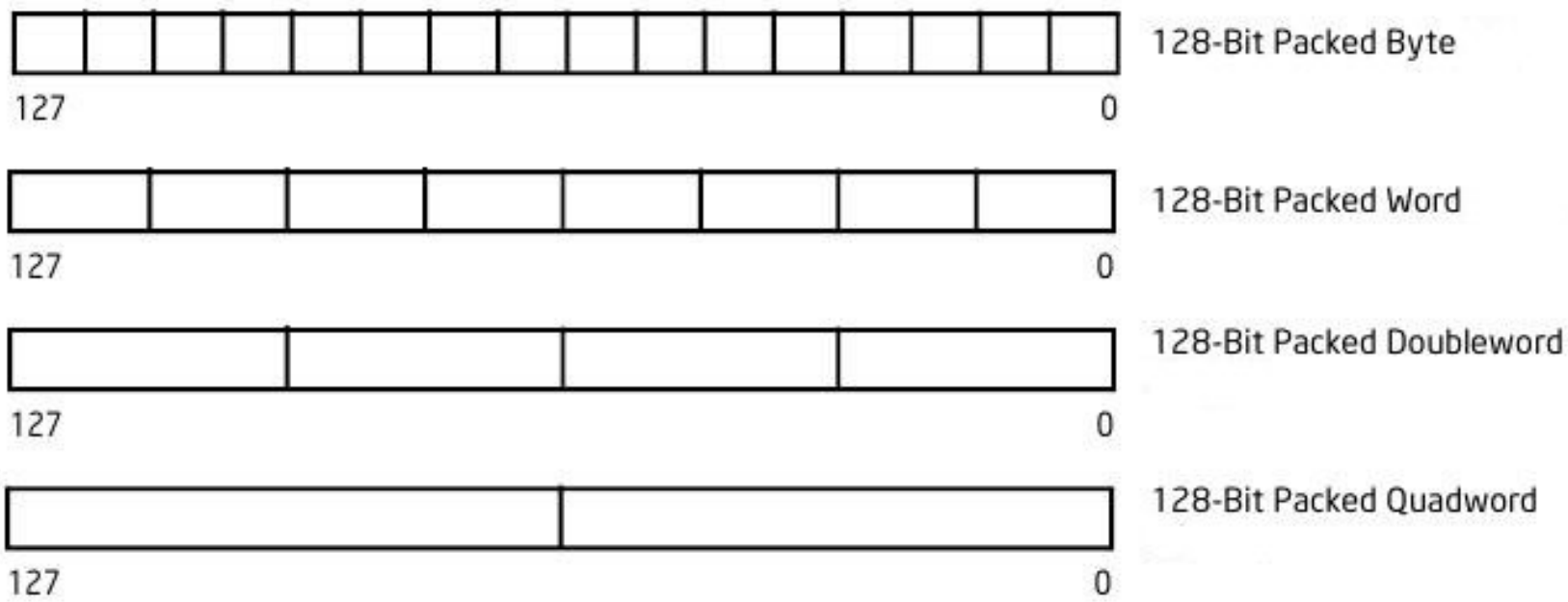
En un registro de **SSE** (**16 bytes**) podemos poner

- **16** datos enteros de **1 byte** (**char**)
- **8** datos enteros de **2 bytes** (**short**)
- **4** datos enteros de **4 bytes** (**int**)
- **2** datos enteros de **8 bytes** (**long long int**)
- **4** datos de punto flotante de **4 bytes** (**float**)
- **2** datos de punto flotante de **8 bytes** (**double**)

Esto se conoce como *empaquetamiento* (guardar más de un dato en un mismo registro), que difiere de *empaquetado* (lo veremos más adelante)

Nota: Tenemos instrucciones que trabajan de forma empaquetada (varios datos a la vez, ej: `ADDPS`) o escalar (sólo uno, ej: `ADDSS`)

Empaquetamiento



¿Cómo sabemos qué tipo de dato tenemos dentro de un registro?

No hay manera de hacerlo desde las instrucciones

Por ende es necesario llevar un seguimiento del tamaño y el tipo de los datos mientras se está escribiendo el código

Ojo: Aún así las instrucciones operarán interpretando los datos como suelen hacerlo

¿Cómo operamos sobre esos registros?

El set de instrucciones **SSE** brinda operaciones de

- Movimiento de datos
- Aritméticas
- Lógicas
- de Comparación
- Trascendentales

Se caracterizan según

- **tipo** de dato con el que operan
- **tamaño** de dato con el que operan

Instrucciones: ¿Cómo leemos el set de instrucciones?

Operaciones de movimiento de datos

Hay dos opciones: movimientos de datos **alineados** y movimientos **desalineados** (al mover datos desde/hacia memoria)

Para movimientos de 128 bits (reg-reg/mem-reg/reg-mem):

- `MOVDQU`: MOV+DQ (double quad word)+U (**U**naligned)
- `MOVDQA`: MOV+DQ (double quad word)+A (**A**ligned)

Para movimientos de menos de 128 bits con regs XMM

- `MOVD`: mueve un double-word a la parte baja de un reg XMM
- `MOVQ`: mueve un quad-word a la parte baja de un reg XMM
- `MOVSS`: mueve un float a la parte baja de un reg XMM
- `MOVSD`: mueve un double a la parte baja de un reg XMM

Nota 1: Hay más instrucciones, estas son sólo algunas

Nota 2: Entre enteros y números de punto flotante el movimiento se efectúa igual aunque varía el funcionamiento a nivel microarquitectura

Instrucciones: ¿Cómo leemos el set de instrucciones?

Operaciones aritméticas (La gran mayoría sigue una regla)

Para enteros: comienzan con `P`, luego el nombre de la *operación* y terminan con el *tamaño* del dato

Ejemplo:

- `PADDB`: suma de a **B**yte
- `PADDW`: suma de a **W**ord
- `PADDD`: suma de a **D**oubleword
- `PADDQ`: suma de a **Q**uadword

Para punto flotante: nombre de la *operación*, *modo* de operación y *tamaño* del dato

Ejemplo:

- `ADDPS`: suma de a **F**loat
- `ADDPD`: suma de a **D**ouble

La `P` proviene de `P`acked, podría ser `S` de `S`calar

Ejercicio

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: Paso a paso

Veamos cómo podemos encarar el ejercicio:

- Tenemos 2 vectores de n datos de tipo float (32 bits) y n es múltiplo de 4
- ¿Cuántos datos podemos guardar a la vez en un registro XMM? $\rightarrow$ 4
- Esto nos dice que vamos a poder procesar como mucho de a 4 datos por instrucción
- ¿Qué instrucciones podemos usar para procesar el producto interno?
  Recordemos la notación de las instrucciones aritméticas y veamos en el manual qué tenemos disponible $\rightarrow$ `MULPS`, `ADDPS`
- Con esto podemos empezar a escribir una rutina que recorra ambos vectores y vaya almacenando el resultado en un registro XMM

Ejercicio: Solución (parte 1)

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

push rbp
mov rbp, rsp
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
    add rdi, 16         ; Avanzar los punteros
    add rsi, 16
    loop .ciclo
```

Ejercicio: Sigamos caminando

Sigamos con nuestra resolución:

- ¿Qué forma tiene tiene el resultado acumulado hasta ahora?
  Tenemos un registro con 4 paquetes de floats a sumar entre sí
- ¿Qué tipo de instrucciones podemos usar para hacer esto? $\rightarrow$ desplazamiento
- Ok ¿cuáles usamos?
  Revisemos la notación y volvamos al manual $\rightarrow$ `PSRLDQ`
- Con esto último ya estamos en condiciones de almacenar el resultado en un registro y devolverlo

Ejercicio: Solución (parte 2)

```
                        ; Sumar todo
movups xmm6, xmm7       ; xmm6 =  sum3 | sum2 | sum1 | sum0
psrldq xmm6, 8          ; xmm6 =    0  |   0  | sum3 | sum2
addps xmm7, xmm6        ; xmm7 =  .  | .  | s3 + s1 | s2 + s0

movups xmm6, xmm7       ; xmm6 = .  | .  | s3 + s1 | s2 + s0
psrldq xmm6, 4          ; xmm6 = .  | .  |    .    | s3 + s1
addps xmm7, xmm6        ; xmm7 = .  | .  |    .    | sumatoria

movss xmm0, xmm7        ; xmm0 = ... | ... | ... | sumatoria

pop rbp
ret
```

Desempaquetado

Motivación

Supongamos que queremos pasar una imagen a escala de grises.

Una forma de hacerlo es a través de la fórmula:

$$f(r, g, b) = \frac{1}{4} \cdot (r + 2g + b)$$

¿Podría haber algún problema? **Posible overflow en la suma**

- Para no perder información en los cálculos, es necesario manejar los resultados intermedios en un tipo de dato de mayor rango (precisión)
- Nuestro tipo de dato es **byte** por lo que deberíamos operar con datos en tamaño **word**
- ¿Cómo hacemos esto?
  Utilizando las instrucciones de desempaquetado

Desempaquetado: `punpck{l,h}{bw,wd,dq,qdq}`

Estas instrucciones se clasifican según la parte del registro a desempaquetar y el tamaño:

- `l`, `h` → parte baja (**l**ow) o alta (**h**igh)
- `bw`, `wd`, `dq`, `qdq` → de **b**yte a **w**ord, de **w**ord a **d**word, etc. . .

No obstante, tras operar necesitamos recuperar el tamaño de dato original para almacenar el resultado

¿Cómo hacemos esto? Utilizando instrucciones de empaquetado

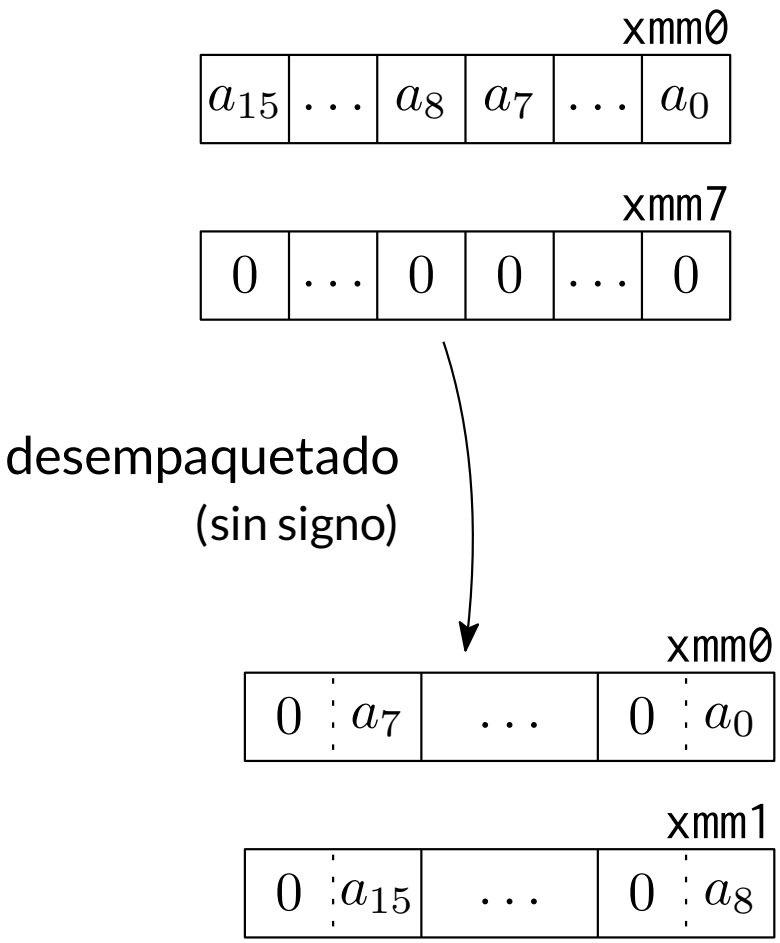
Empaquetado: `pack{ss,us}{wb,dw}`

Estas se clasifican según el signo y el tamaño del dato.
¡Ojo! Los datos a empaquetar tienen saturación

- `ss`, `us` → **s**igned / **u**nsigned, con **s**aturación
- `wb`, `dw` → de **w**ord a **b**yte, de **d**word a **w**ord

Desempaquetado

En código




```
pxor xmm7, xmm7
movdqu xmm1, xmm0
punpcklbw xmm0, xmm7
punpckhbw xmm1, xmm7
```

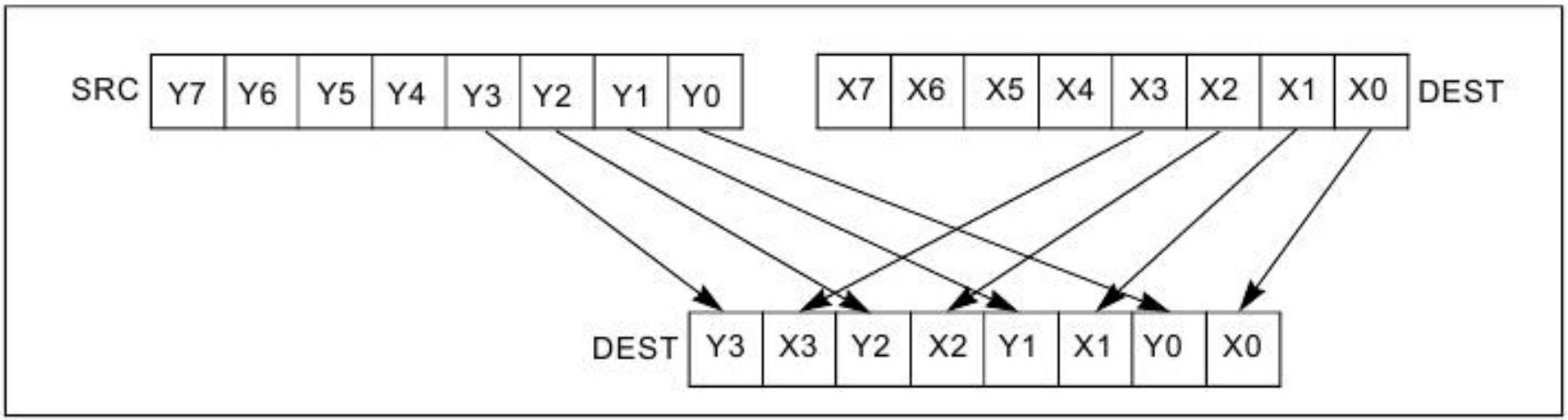
Ahora tenemos los valores originales en tamaño **word**

Notar que se hace interleaving (intercalado) entre la parte alta/baja (según corresponda) de ambos registros de la operación

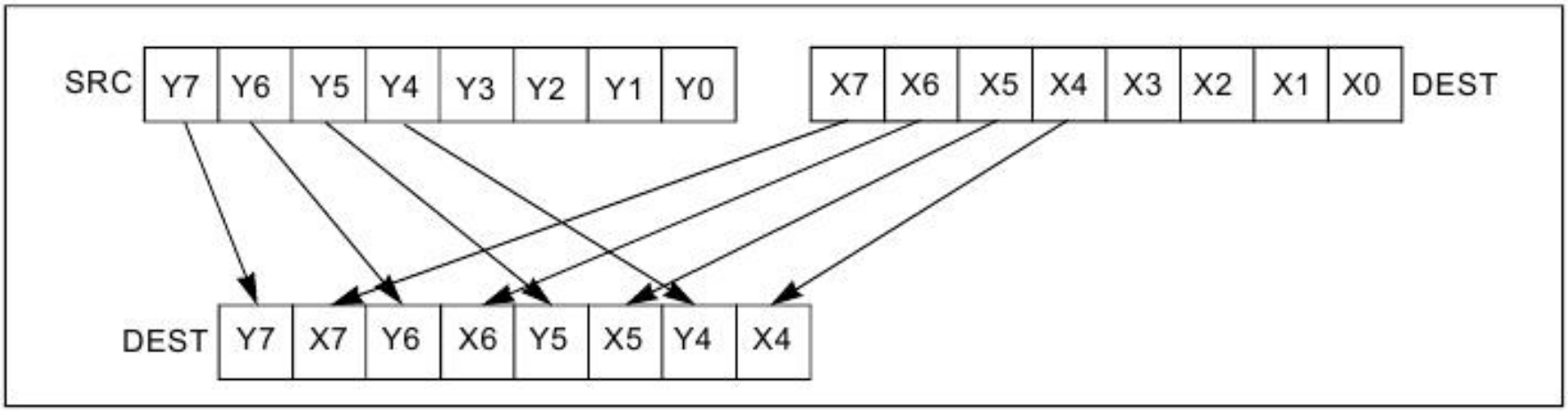
Desempaquetado

En más detalle

punpckLbw



punpckHbw



En SSE también existen instrucciones de comparación, aunque se comportan un poco diferente a las que veníamos usando

Claramente **no se pueden usar saltos condicionales** porque estamos trabajando con muchos datos al mismo tiempo

Entonces **podemos usar máscaras** obtenidas a partir de comparaciones de a paquetes de cierto tamaño

	-7	42	-5	57
$<$	0	0	0	0
	1	0	1	0

Comparación

Ejemplo

Supongamos que estamos trabajando con **words** y queremos saber qué elementos del registro son menores a cero

Datos en `xmm0`

1000	-456	-15	0	100	234	-890	1

```
pxor xmm7, xmm7                 ; xmm7 = 0 | 0 | ... | 0
pcmpgtw xmm7, xmm0              ; xmm7 > xmm0 ?
```

Resultado en `xmm7`

0x0000	0xFFFF	0xFFFF	0x0000	0x0000	0x0000	0xFFFF	0x0000

Es decir, compara **word a word** y si se cumple la condición setea **unos** (0xFFFF, en este caso) en el resultado, o **ceros** de no ser así

Comparación

Usos

Las instrucciones de comparación nos devuelven un registro con unos y ceros según los datos que cumplan con una condición dada

Podemos usarlas para:

- **Extender el signo** de números signados al desempaquetar, teniendo en cuenta que los números negativos tienen sus bits más significativos en uno, y los positivos en cero
  - $-5 = 1011 \rightsquigarrow 1111\,1011$
  - $+5 = 0101 \rightsquigarrow 0000\,0101$
- **Crear una máscara**, y usarla para operar con instrucciones como `PAND`, `POR`, etc. Esta nos va a permitir simular las dos ramas provenientes de un salto condicional

Para extender el signo del registro del ejemplo anterior:

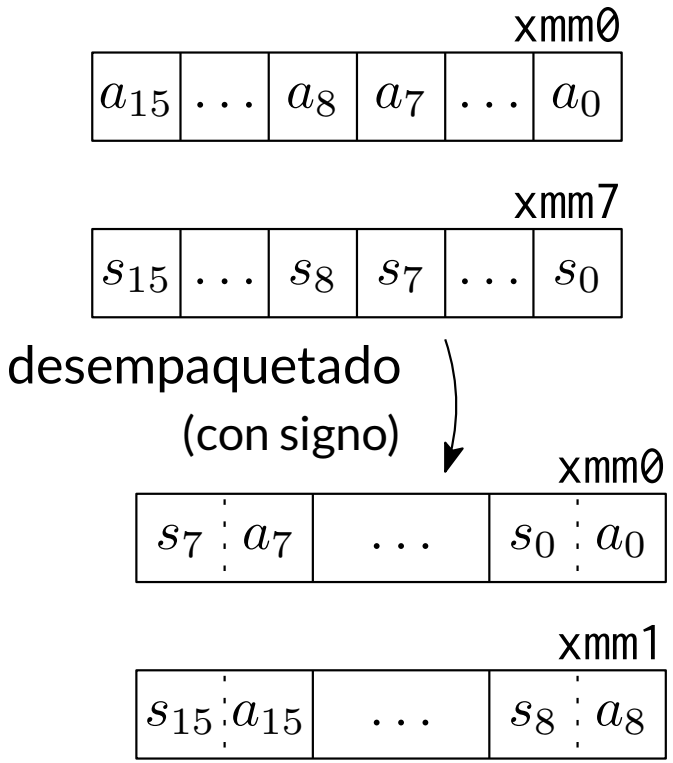
Datos en `xmm0`

1000	-456	-15	0	100	234	-890	1

Si tenemos el resultado de la comparación en `xmm7`

```
movdqu xmm1, xmm0
punpckhwd xmm0, xmm7
punpcklwd xmm1, xmm7
```

Tarea: intentar pronunciar las instrucciones



Cálculo de Padding

¿Qué es el padding?

Anteriormente vimos:

- que el modelo de procesamiento SIMD se usa para procesamiento multimedia, particularmente de imágenes
- que hay instrucciones, como por ejemplo `mov`, que están optimizadas para mover datos alineados en memoria

Supongamos ahora que tenemos la siguiente matriz de floats. Cada uno representa la intensidad de un píxel en una imagen de escala de grises de 3x5:

0.13	0.1	0.13	0.34	0.99
0.14	0.913	0.15	0.36	0.0
0.94	0.15	0.43	0.59	0.9

¿Cómo están los datos al inicio de cada fila de la matriz?
Algunos están alineados y otros no

¿Qué podemos hacer entonces para salvar esta desalineación?

Introducimos un padding al final de cada fila para hacer que la siguiente esté alineada a 16 bytes. De esta manera, la imagen anterior estará representada en memoria como sigue:

0.13	0.1	0.13	0.34	0.99			
0.14	0.913	0.15	0.36	0.0			
0.94	0.15	0.43	0.59	0.9			

Aquí las celdas azules son porciones de memoria de 4 bytes que se utilizan como padding. Esto hace que al cargar los últimos datos de una fila también carguemos estas celdas y tengamos que tratarlas correspondientemente

Cálculo de Padding

Recorriendo la matriz

Al recorrer una matriz con padding, debemos tener en cuenta que:

- La manera en que procesemos el padding, **depende fuertemente de lo que estemos haciendo con la matriz**. Generalmente vamos a descartar los datos a través de, por ejemplo, operaciones de desplazamiento
- **No hay garantías de que el padding contenga 0**. Comúnmente contiene datos basura por lo que al procesarlos estaríamos usando datos no inicializados

Al tener valores no inicializados Valgrind podría dar un error, el cual puede ser ignorado

Procesamiento vectorial, ¿cómo nos podemos equivocar?

Algunos de los errores más comunes son:

- Usar instrucciones de enteros para punto flotante y viceversa
- No desempaquetar enteros y causar un overflow (o hacerlo cuando no hace falta)
- Usar instrucciones que no existen (más que nada en el parcial)
- Usar branches (ramas condicionales) en vez de máscaras
- Calcular mal el padding al utilizar matrices

Siendo así, recomendamos:

- Llevar un seguimiento del tipo y tamaño de los datos en cada sección del código (comentarios, esquemas)
- Prestar atención a la cantidad de datos procesados por cada instrucción y su efecto en los registros operando (ver manual)
- Crear máscaras relacionadas con el comportamiento de cada rama de una condición

Bibliografía

- Segundo tomo del manual de Intel:
  - Oficial:
    `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`
  - No oficial pero en formato más amigable:
    `https://www.felixcloutier.com/x86/index.html`
- Reseña general sobre SIMD:
  `https://en.wikipedia.org/wiki/SIMD`

¿Preguntas?