

Modelo de Ejecución SIMD en IA-32

Alejandro Furfaro - David González Márquez

31 de agosto de 2023

# Agenda



- 1 Fundamentos
  - Sobre Fourier, Lagrange, Laplace y los grupos de Whatsapp
- 2 Procesamiento de Señales digitales
  - Digitalización de la señal
  - Arquitecturas de Procesamiento de una señal digital
- 3 Modelo de ejecución SIMD
  - Un modelo de paralelización
- 4 Implementaciones SIMD en x86
  - Arquitectura completa
  - Números Reales
  - Formatos de Punto Fijo
  - Formatos de Punto Flotante
  - Codificación de Números Reales
- 5 Instrucciones
  - Transferencias (las mas comunes)
  - Aritmética en algoritmos DSP
  - Instrucciones de punto flotante
  - Instrucciones para manejo de enteros para SSEn
  - Instrucciones para manejo de cacheabilidad
- 6 Advanced Vector Extensions
  - Preliminar
  - Detección de las facilidades AVX

## 1 Fundamentos

- Sobre Fourier, Lagrange, Laplace y los grupos de Whatsapp

## 2 Procesamiento de Señales digitales

## 3 Modelo de ejecución SIMD

## 4 Implementaciones SIMD en x86

## 5 Instrucciones

## 6 Advanced Vector Extensions

# Épocas en las que no había Whatsapp



Jean-Baptiste Joseph Fourier, Joseph-Louis Lagrange. y Pierre-Simon Laplace...

# Serie de Fourier

# Serie de Fourier

- Partió de demostrar que las funciones:

$$\operatorname{sen}(n \cdot x), \cos(n \cdot x), \forall n \in \text{entero}, \quad (1)$$

son funciones ortogonales, es decir, permiten calcular cualquier otra función en términos de este par de funciones, como una sucesión de cálculos que pueden expresarse genéricamente como:

# Serie de Fourier

- Partió de demostrar que las funciones:

$$\sin(n \cdot x), \cos(n \cdot x), \forall n \in \text{entero}, \quad (1)$$

son funciones ortogonales, es decir, permiten calcular cualquier otra función en términos de este par de funciones, como una sucesión de cálculos que pueden expresarse genéricamente como:

$$f(t) \approx \frac{a_0}{2} + \sum_{n=1}^{\infty} [a_n \cdot \cos\left(\frac{2n\pi}{T} \cdot t\right) + b_n \cdot \sin\left(\frac{2n\pi}{T} \cdot t\right)], \quad (2)$$

# Serie de Fourier

- Partió de demostrar que las funciones:

$$\sin(n \cdot x), \cos(n \cdot x), \forall n \in \text{entero}, \quad (1)$$

son funciones ortogonales, es decir, permiten calcular cualquier otra función en términos de este par de funciones, como una sucesión de cálculos que pueden expresarse genéricamente como:

$$f(t) \approx \frac{a_0}{2} + \sum_{n=1}^{\infty} [a_n \cdot \cos\left(\frac{2n\pi}{T} \cdot t\right) + b_n \cdot \sin\left(\frac{2n\pi}{T} \cdot t\right)], \quad (2)$$

- Donde:

$$a_0 = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot dt, a_n = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot \cos(n\omega_0 t) \cdot dt, b_n = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot \sin(n\omega_0 t) \cdot dt. \quad (3)$$

# Serie de Fourier

- Partió de demostrar que las funciones:

$$\sin(n \cdot x), \cos(n \cdot x), \forall n \in \text{entero}, \quad (1)$$

son funciones ortogonales, es decir, permiten calcular cualquier otra función en términos de este par de funciones, como una sucesión de cálculos que pueden expresarse genéricamente como:

$$f(t) \approx \frac{a_0}{2} + \sum_{n=1}^{\infty} [a_n \cdot \cos\left(\frac{2n\pi}{T} \cdot t\right) + b_n \cdot \sin\left(\frac{2n\pi}{T} \cdot t\right)], \quad (2)$$

- Donde:

$$a_0 = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot dt, a_n = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot \cos(n\omega_0 t) \cdot dt, b_n = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot \sin(n\omega_0 t) \cdot dt. \quad (3)$$

- Su demostración es larga y compleja, y no es el objetivo de este curso.

# Serie de Fourier: Señal rectangular

La función rectangular es una función discontinua definida mediante la siguiente expresión.

$$f(t) = \begin{cases} -1 & -1 \leq t \leq 0 \\ 1 & 0 \leq t \leq 1 \end{cases}$$

## Serie de Fourier: Señal rectangular

La función rectangular es una función discontinua definida mediante la siguiente expresión.

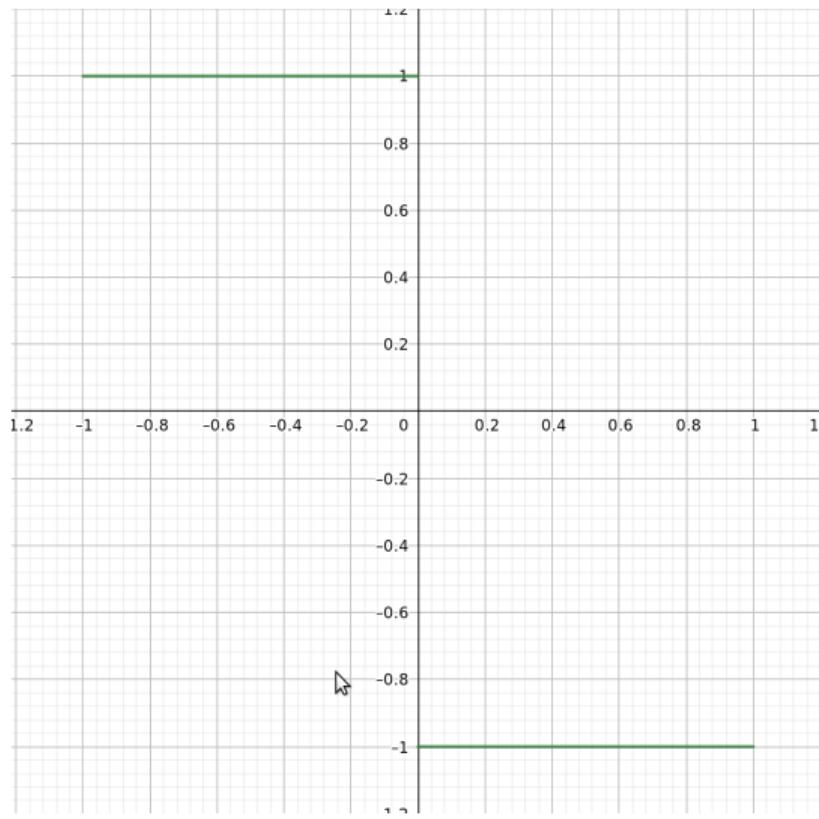
$$f(t) = \begin{cases} -1 & -1 \leq t \leq 0 \\ 1 & 0 \leq t \leq 1 \end{cases}$$

O mas genéricamente:

$$f(t) = \begin{cases} -1 & -T/2 \leq t \leq 0 \\ 1 & 0 \leq t \leq T/2 \end{cases}; \text{ con } T = 2\pi/\omega_0$$

Siendo T, el período de la función, y  $\omega_0$  la frecuencia angular medida en radianes.

Gráficamente...



# Función Rectangular: Coeficiente $a_0$ de Fourier

$$a_0 = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot dt$$

# Función Rectangular: Coeficiente $a_0$ de Fourier

$$a_0 = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot dt$$

$$a_0 = \frac{2}{T} \left[ \int_{-\frac{T}{2}}^0 -dt + \int_{-0}^{\frac{T}{2}} dt \right] = \frac{2}{T} \cdot \left[ -t \Big|_{-\frac{T}{2}}^0 + t \Big|_0^{\frac{T}{2}} \right]$$

# Función Rectangular: Coeficiente $a_0$ de Fourier

$$a_0 = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot dt$$

$$a_0 = \frac{2}{T} \left[ \int_{-\frac{T}{2}}^0 -dt + \int_{-0}^{\frac{T}{2}} dt \right] = \frac{2}{T} \cdot \left[ -t \Big|_{-\frac{T}{2}}^0 + t \Big|_0^{\frac{T}{2}} \right]$$

$$a_0 = \frac{2}{T} \left[ (0 \cdot (-1)) - \left( -\frac{T}{2} \cdot (-1) \right) + \left( \frac{T}{2} \right) + (0) \right]$$

# Función Rectangular: Coeficiente $a_0$ de Fourier

$$a_0 = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot dt$$

$$a_0 = \frac{2}{T} \left[ \int_{-\frac{T}{2}}^0 -dt + \int_{-0}^{\frac{T}{2}} dt \right] = \frac{2}{T} \cdot \left[ -t \Big|_{-\frac{T}{2}}^0 + t \Big|_0^{\frac{T}{2}} \right]$$

$$a_0 = \frac{2}{T} \left[ (0 \cdot (-1)) - \left( -\frac{T}{2} \cdot (-1) \right) + \left( \frac{T}{2} \right) + (0) \right]$$

$$a_0 = 0$$

(4)

# Función Rectangular: Coeficientes $a_n$ de Fourier

$$a_n = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot \cos(n\omega_0 t) \cdot dt$$

# Función Rectangular: Coeficientes $a_n$ de Fourier

$$a_n = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot \cos(n\omega_0 t) \cdot dt$$

$$a_n = \frac{2}{T} \cdot \left[ \int_{-\frac{T}{2}}^0 (-1) \cos(n\omega_0 t) \cdot dt + \int_{-0}^{\frac{T}{2}} (1) \cos(n\omega_0 t) \cdot dt \right]$$

# Función Rectangular: Coeficientes $a_n$ de Fourier

$$a_n = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot \cos(n\omega_0 t) \cdot dt$$

$$a_n = \frac{2}{T} \cdot \left[ \int_{-\frac{T}{2}}^0 (-1) \cos(n\omega_0 t) \cdot dt + \int_{-0}^{\frac{T}{2}} (1) \cos(n\omega_0 t) \cdot dt \right]$$

$$a_n = \frac{2}{T} \cdot \left[ -\frac{1}{n\omega_0} \cdot \sin(n\omega_0 t) \Big|_{\frac{T}{2}}^0 + \frac{1}{n\omega_0} \cdot \sin(n\omega_0 t) \Big|_0^{\frac{T}{2}} \right]$$

# Función Rectangular: Coeficientes $a_n$ de Fourier

$$a_n = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot \cos(n\omega_0 t) \cdot dt$$

$$a_n = \frac{2}{T} \cdot \left[ \int_{-\frac{T}{2}}^0 (-1) \cos(n\omega_0 t) \cdot dt + \int_{-0}^{\frac{T}{2}} (1) \cos(n\omega_0 t) \cdot dt \right]$$

$$a_n = \frac{2}{T} \cdot \left[ -\frac{1}{n\omega_0} \cdot \sin(n\omega_0 t) \Big|_{\frac{T}{2}}^0 + \frac{1}{n\omega_0} \cdot \sin(n\omega_0 t) \Big|_0^{\frac{T}{2}} \right]$$

$a_n = 0, \forall n \neq 0$

(5)

# Función Rectangular: Coeficientes $b_n$ de Fourier

$$b_n = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot \sin(n\omega_0 t) \cdot dt$$

# Función Rectangular: Coeficientes $b_n$ de Fourier

$$b_n = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot \sin(n\omega_0 t) \cdot dt$$

$$b_n = \frac{2}{T} \left[ \int_{-\frac{T}{2}}^0 (-1) \sin(n\omega_0 t) \cdot dt + \int_0^{\frac{T}{2}} (1) \sin(n\omega_0 t) \cdot dt \right] = \frac{2}{T} \cdot \left[ \frac{1}{n\omega_0} \cdot \cos(n\omega_0 t) \Big|_{-\frac{T}{2}}^0 - \frac{1}{n\omega_0} \cdot \cos(n\omega_0 t) \Big|_0^{\frac{T}{2}} \right]$$

# Función Rectangular: Coeficientes $b_n$ de Fourier

$$b_n = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot \sin(n\omega_0 t) \cdot dt$$

$$b_n = \frac{2}{T} \left[ \int_{-\frac{T}{2}}^0 (-1) \sin(n\omega_0 t) \cdot dt + \int_0^{\frac{T}{2}} (1) \sin(n\omega_0 t) \cdot dt \right] = \frac{2}{T} \cdot \left[ \frac{1}{n\omega_0} \cdot \cos(n\omega_0 t) \Big|_{-\frac{T}{2}}^0 - \frac{1}{n\omega_0} \cdot \cos(n\omega_0 t) \Big|_0^{\frac{T}{2}} \right]$$

$$b_n = \frac{2}{\frac{2\pi}{\omega_0}} \cdot \frac{1}{n\omega_0} \cdot \left[ \cos 0 + \cos(n\omega_0 \frac{T}{2}) - \left( \cos(n\omega_0 \frac{T}{2}) - \cos 0 \right) \right]$$

# Función Rectangular: Coeficientes $b_n$ de Fourier

$$b_n = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot \sin(n\omega_0 t) \cdot dt$$

$$b_n = \frac{2}{T} \left[ \int_{-\frac{T}{2}}^0 (-1) \sin(n\omega_0 t) \cdot dt + \int_0^{\frac{T}{2}} (1) \sin(n\omega_0 t) \cdot dt \right] = \frac{2}{T} \cdot \left[ \frac{1}{n\omega_0} \cdot \cos(n\omega_0 t) \Big|_{-\frac{T}{2}}^0 - \frac{1}{n\omega_0} \cdot \cos(n\omega_0 t) \Big|_0^{\frac{T}{2}} \right]$$

$$b_n = \frac{2}{\frac{2\pi}{\omega_0}} \cdot \frac{1}{n\omega_0} \cdot \left[ \cos 0 + \cos(n\omega_0 \frac{T}{2}) - \left( \cos(n\omega_0 \frac{T}{2}) - \cos 0 \right) \right]$$

$$b_n = \frac{2}{n\pi} \cdot \left[ (1 - \cos(n\pi)) - (\cos(n\pi) - 1) \right]$$

# Función Rectangular: Coeficientes $b_n$ de Fourier

$$b_n = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cdot \sin(n\omega_0 t) \cdot dt$$

$$b_n = \frac{2}{T} \left[ \int_{-\frac{T}{2}}^0 (-1) \sin(n\omega_0 t) \cdot dt + \int_0^{\frac{T}{2}} (1) \sin(n\omega_0 t) \cdot dt \right] = \frac{2}{T} \cdot \left[ \frac{1}{n\omega_0} \cdot \cos(n\omega_0 t) \Big|_{-\frac{T}{2}}^0 - \frac{1}{n\omega_0} \cdot \cos(n\omega_0 t) \Big|_0^{\frac{T}{2}} \right]$$

$$b_n = \frac{2}{\frac{2\pi}{\omega_0}} \cdot \frac{1}{n\omega_0} \cdot \left[ \cos 0 + \cos(n\omega_0 \frac{T}{2}) - \left( \cos(n\omega_0 \frac{T}{2}) - \cos 0 \right) \right]$$

$$b_n = \frac{2}{n\pi} \cdot \left[ (1 - \cos(n\pi)) - (\cos(n\pi) - 1) \right]$$

$$b_n = \frac{2}{n\pi} [1 - (-1)^n], \forall n \neq 0$$

(6)

# Resultado

- La ecuación (6) es distinta de cero para los valores de  $n$  impares. Por lo tanto la función descompuesta en serie de Fourier queda del siguiente modo:

# Resultado

- La ecuación (6) es distinta de cero para los valores de  $n$  impares. Por lo tanto la función descompuesta en serie de Fourier queda del siguiente modo:

$$\begin{aligned}f(t) = & \frac{2}{1\pi} [1 - (-1)^1] \cdot \operatorname{sen}(1 \cdot \omega_0 t) + \frac{2}{3\pi} [1 - (-1)^3] \cdot \operatorname{sen}(3 \cdot \omega_0 t) + \\& + \frac{2}{5\pi} [1 - (-1)^5] \cdot \operatorname{sen}(5 \cdot \omega_0 t) + \frac{2}{7\pi} [1 - (-1)^7] \cdot \operatorname{sen}(7 \cdot \omega_0 t) + \dots\end{aligned}$$

# Resultado

- La ecuación (6) es distinta de cero para los valores de  $n$  impares. Por lo tanto la función descompuesta en serie de Fourier queda del siguiente modo:

$$\begin{aligned}f(t) &= \frac{2}{1\pi} [1 - (-1)^1] \cdot \operatorname{sen}(1 \cdot \omega_0 t) + \frac{2}{3\pi} [1 - (-1)^3] \cdot \operatorname{sen}(3 \cdot \omega_0 t) + \\&\quad + \frac{2}{5\pi} [1 - (-1)^5] \cdot \operatorname{sen}(5 \cdot \omega_0 t) + \frac{2}{7\pi} [1 - (-1)^7] \cdot \operatorname{sen}(7 \cdot \omega_0 t) + \dots \\f(t) &= \frac{2}{1\pi} [2] \cdot \operatorname{sen}(1 \cdot \omega_0 t) + \frac{2}{3\pi} [2] \cdot \operatorname{sen}(3 \cdot \omega_0 t) + \frac{2}{5\pi} [2] \cdot \operatorname{sen}(5 \cdot \omega_0 t) + \\&\quad + \frac{2}{7\pi} [2] \cdot \operatorname{sen}(7 \cdot \omega_0 t) + \dots\end{aligned}$$

# Resultado

- La ecuación (6) es distinta de cero para los valores de  $n$  impares. Por lo tanto la función descompuesta en serie de Fourier queda del siguiente modo:

$$\begin{aligned}f(t) &= \frac{2}{1\pi} [1 - (-1)^1] \cdot \operatorname{sen}(1 \cdot \omega_0 t) + \frac{2}{3\pi} [1 - (-1)^3] \cdot \operatorname{sen}(3 \cdot \omega_0 t) + \\&\quad + \frac{2}{5\pi} [1 - (-1)^5] \cdot \operatorname{sen}(5 \cdot \omega_0 t) + \frac{2}{7\pi} [1 - (-1)^7] \cdot \operatorname{sen}(7 \cdot \omega_0 t) + \dots\end{aligned}$$

$$\begin{aligned}f(t) &= \frac{2}{1\pi} [2] \cdot \operatorname{sen}(1 \cdot \omega_0 t) + \frac{2}{3\pi} [2] \cdot \operatorname{sen}(3 \cdot \omega_0 t) + \frac{2}{5\pi} [2] \cdot \operatorname{sen}(5 \cdot \omega_0 t) + \\&\quad + \frac{2}{7\pi} [2] \cdot \operatorname{sen}(7 \cdot \omega_0 t) + \dots\end{aligned}$$

$$f(t) = \frac{4}{\pi} \left[ \operatorname{sen}(\omega_0 t) + \frac{1}{3} \cdot \operatorname{sen}(3 \cdot \omega_0 t) + \frac{1}{5} \cdot \operatorname{sen}(5 \cdot \omega_0 t) + \frac{1}{7} \cdot \operatorname{sen}(7 \cdot \omega_0 t) + \dots \right]$$

# Resultado

- La ecuación (6) es distinta de cero para los valores de  $n$  impares. Por lo tanto la función descompuesta en serie de Fourier queda del siguiente modo:

$$\begin{aligned}
 f(t) &= \frac{2}{1\pi} [1 - (-1)^1] \cdot \operatorname{sen}(1 \cdot \omega_0 t) + \frac{2}{3\pi} [1 - (-1)^3] \cdot \operatorname{sen}(3 \cdot \omega_0 t) + \\
 &\quad + \frac{2}{5\pi} [1 - (-1)^5] \cdot \operatorname{sen}(5 \cdot \omega_0 t) + \frac{2}{7\pi} [1 - (-1)^7] \cdot \operatorname{sen}(7 \cdot \omega_0 t) + \dots \\
 f(t) &= \frac{2}{1\pi} [2] \cdot \operatorname{sen}(1 \cdot \omega_0 t) + \frac{2}{3\pi} [2] \cdot \operatorname{sen}(3 \cdot \omega_0 t) + \frac{2}{5\pi} [2] \cdot \operatorname{sen}(5 \cdot \omega_0 t) + \\
 &\quad + \frac{2}{7\pi} [2] \cdot \operatorname{sen}(7 \cdot \omega_0 t) + \dots \\
 f(t) &= \frac{4}{\pi} \left[ \operatorname{sen}(\omega_0 t) + \frac{1}{3} \cdot \operatorname{sen}(3 \cdot \omega_0 t) + \frac{1}{5} \cdot \operatorname{sen}(5 \cdot \omega_0 t) + \frac{1}{7} \cdot \operatorname{sen}(7 \cdot \omega_0 t) + \dots \right]
 \end{aligned}$$

Donde,  $\omega_0 = \frac{2 \cdot \pi}{T}$ , y  $T = (1) - (-1) = 2 \Rightarrow \omega_0 = \pi$ .

# Resultado

- La ecuación (6) es distinta de cero para los valores de  $n$  impares. Por lo tanto la función descompuesta en serie de Fourier queda del siguiente modo:

$$f(t) = \frac{2}{1\pi} [1 - (-1)^1] \cdot \operatorname{sen}(1 \cdot \omega_0 t) + \frac{2}{3\pi} [1 - (-1)^3] \cdot \operatorname{sen}(3 \cdot \omega_0 t) + \\ + \frac{2}{5\pi} [1 - (-1)^5] \cdot \operatorname{sen}(5 \cdot \omega_0 t) + \frac{2}{7\pi} [1 - (-1)^7] \cdot \operatorname{sen}(7 \cdot \omega_0 t) + \dots$$

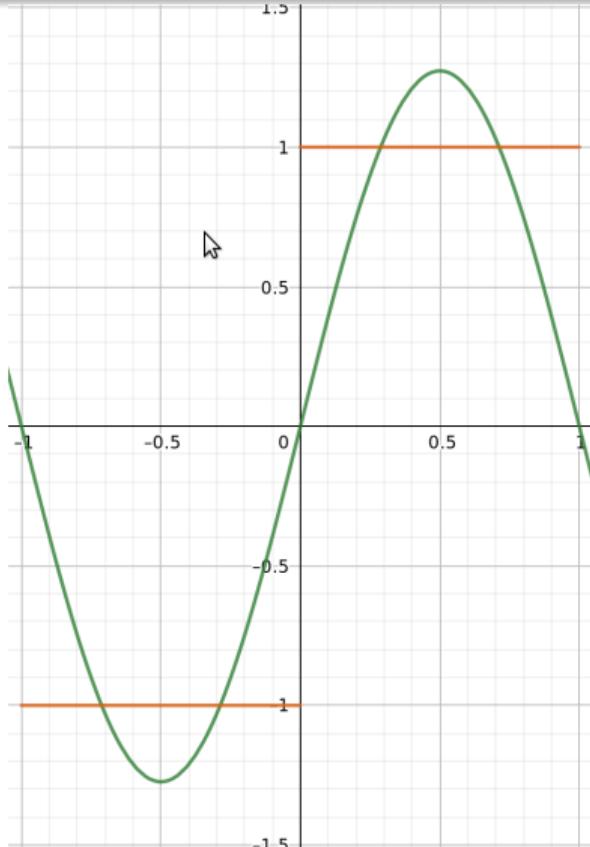
$$f(t) = \frac{2}{1\pi} [2] \cdot \operatorname{sen}(1 \cdot \omega_0 t) + \frac{2}{3\pi} [2] \cdot \operatorname{sen}(3 \cdot \omega_0 t) + \frac{2}{5\pi} [2] \cdot \operatorname{sen}(5 \cdot \omega_0 t) + \\ + \frac{2}{7\pi} [2] \cdot \operatorname{sen}(7 \cdot \omega_0 t) + \dots$$

$$f(t) = \frac{4}{\pi} \left[ \operatorname{sen}(\omega_0 t) + \frac{1}{3} \cdot \operatorname{sen}(3 \cdot \omega_0 t) + \frac{1}{5} \cdot \operatorname{sen}(5 \cdot \omega_0 t) + \frac{1}{7} \cdot \operatorname{sen}(7 \cdot \omega_0 t) + \dots \right]$$

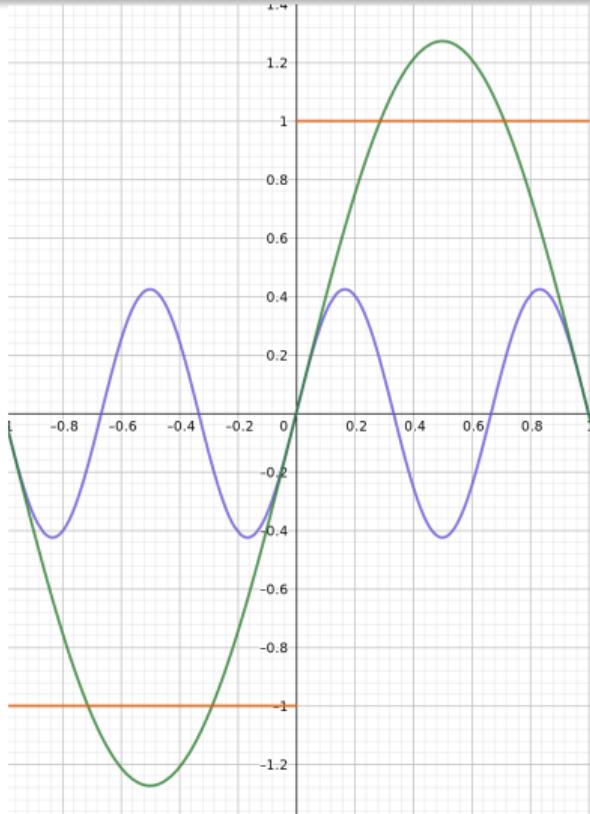
Donde,  $\omega_0 = \frac{2 \cdot \pi}{T}$ , y  $T = (1) - (-1) = 2 \Rightarrow \omega_0 = \pi$ .

$$f(t) = \frac{4}{\pi} \left[ \operatorname{sen}(\pi \cdot t) + \frac{1}{3} \cdot \operatorname{sen}(3\pi \cdot t) + \frac{1}{5} \cdot \operatorname{sen}(5\pi \cdot t) + \frac{1}{7} \cdot \operatorname{sen}(7\pi \cdot t) + \dots \right] \quad (7)$$

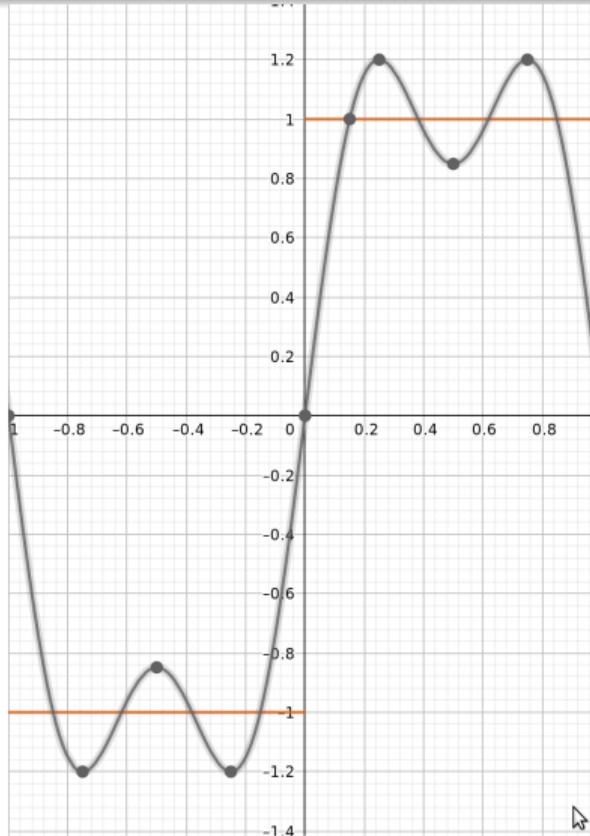
# Aproximando con solo la fundamental...



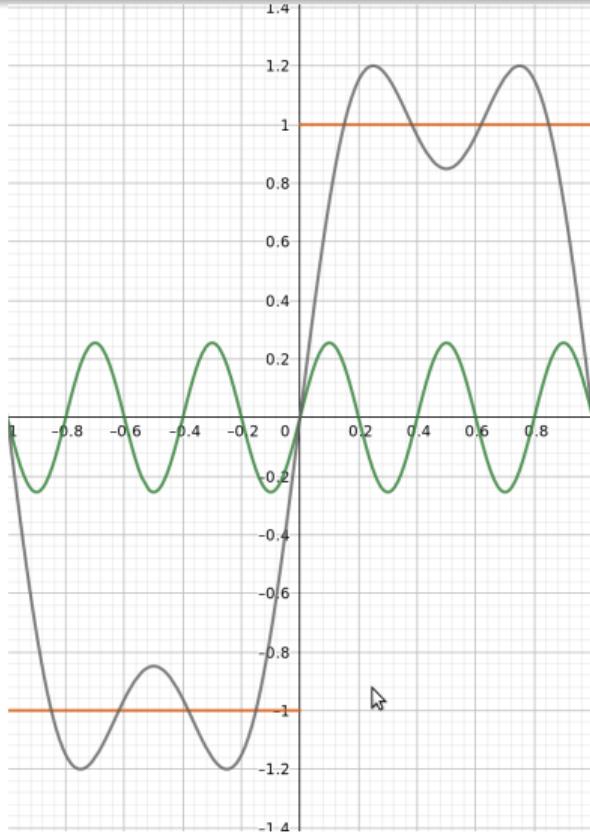
# Aproximando con la fundamental y la 3er. Armónica



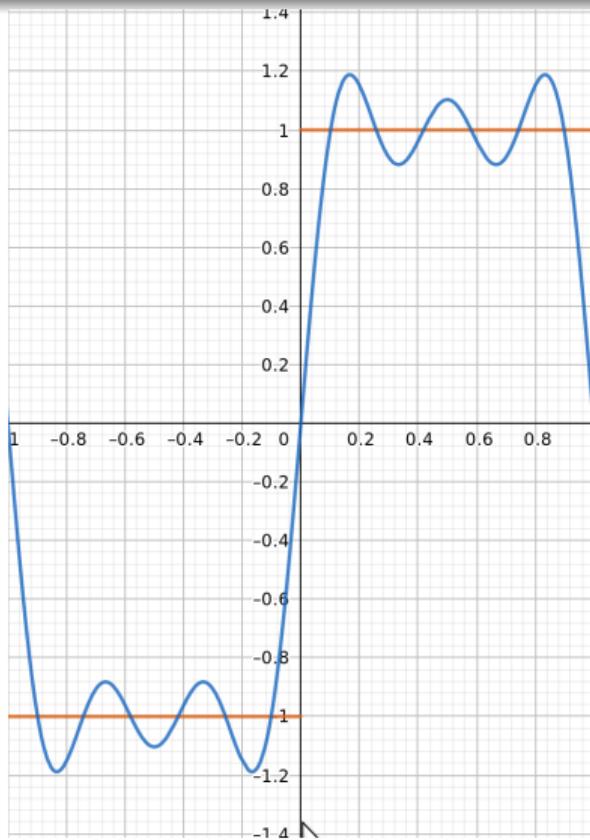
# Sumando la fundamental y la 3er. Armónica



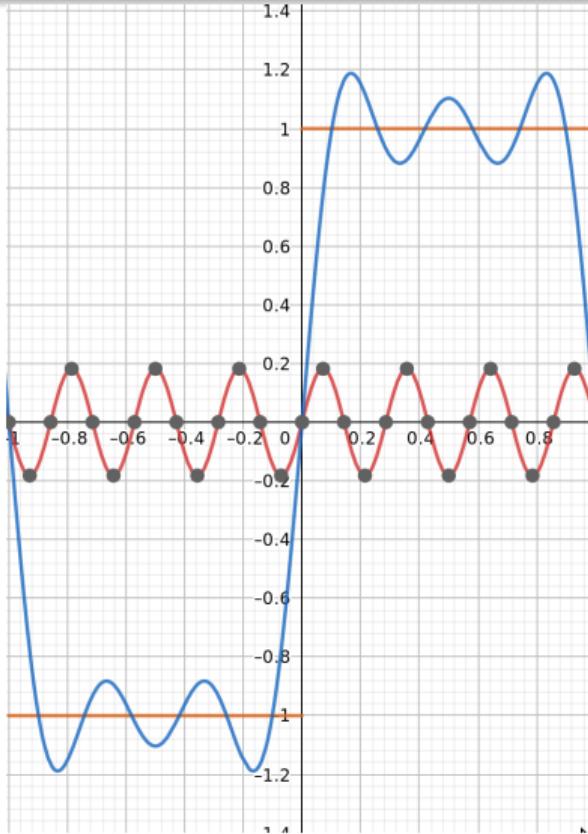
# Agregando la 5ta. Armónica



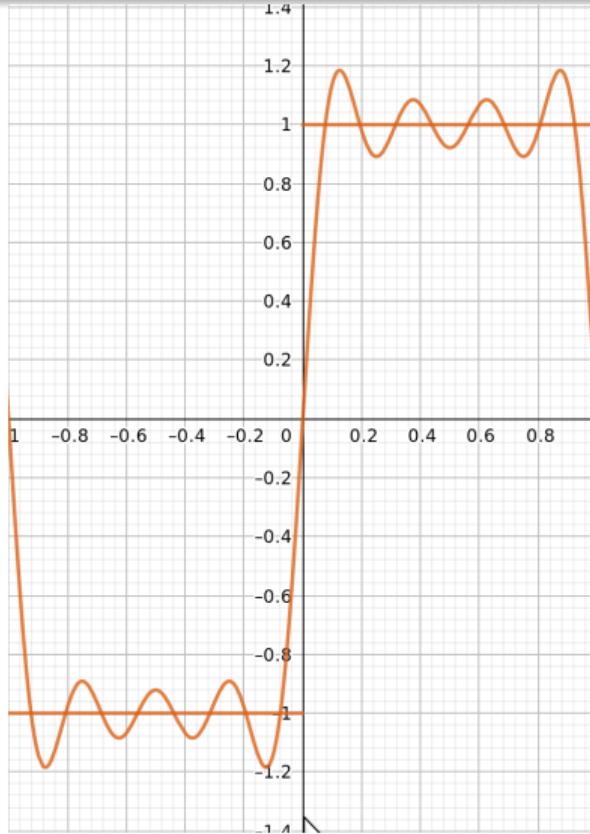
# Sumando la 5ta. Armónica



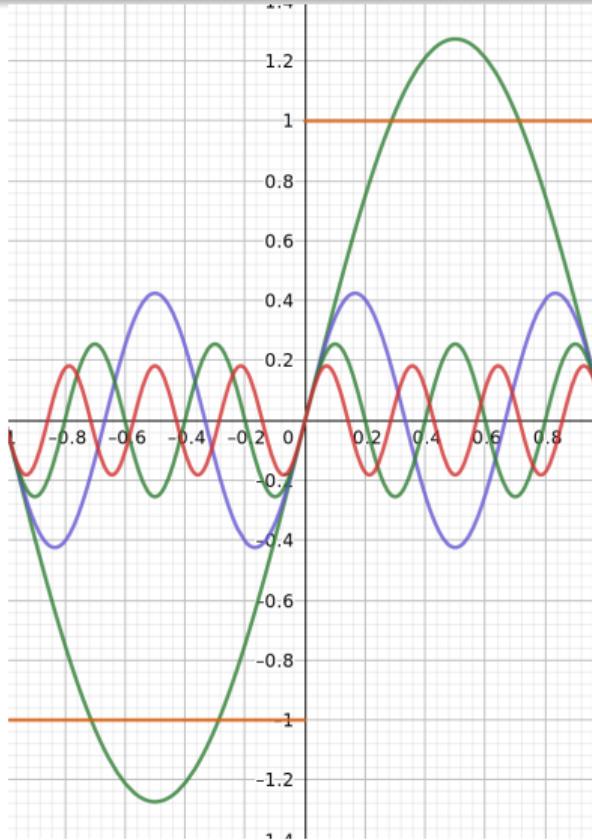
# Agregando la 7ma. Armónica



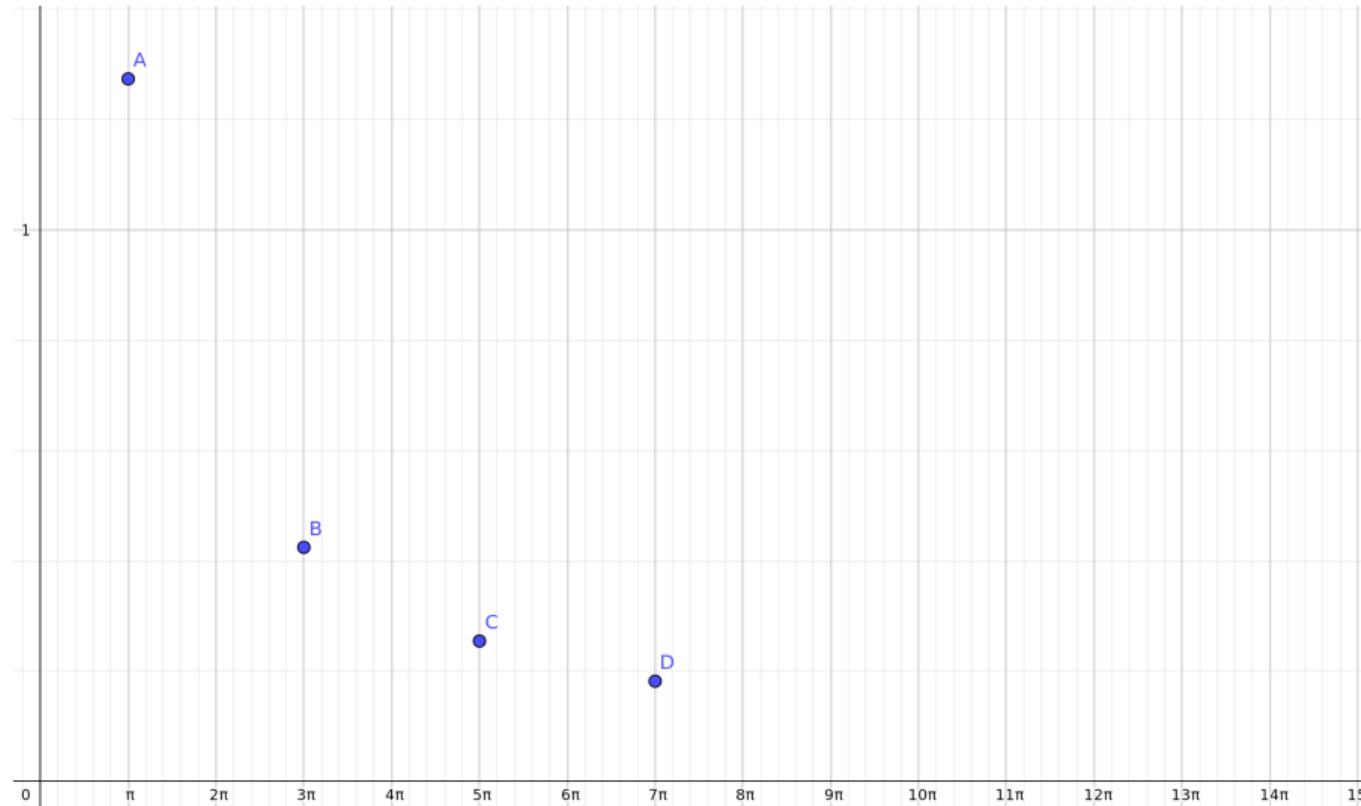
# Sumando la 7ma. Armónica



# Representación en el dominio del tiempo



# Representación en el dominio de la frecuencia



# Observaciones

# Observaciones

- Permite descomponer cualquier señal periódica en señales muy simples de ser analizadas: *seno* y *coseno*.

# Observaciones

- Permite descomponer cualquier señal periódica en señales muy simples de ser analizadas: *seno* y *coseno*.
- Flexibilidad en el requisito de periodicidad: una grabación de audio es finalmente una secuencia sonidos en un tiempo finito, de modo que repetirla infinitamente la convierte en una señal periódica.

# Observaciones

- Permite descomponer cualquier señal periódica en señales muy simples de ser analizadas: *seno* y *coseno*.
- Flexibilidad en el requisito de periodicidad: una grabación de audio es finalmente una secuencia de sonidos en un tiempo finito, de modo que repetirla infinitamente la convierte en una señal periódica.
- Pasamos de representar una señal continua en el dominio del tiempo en una señal discreta en el dominio de la frecuencia.

# Observaciones

- Permite descomponer cualquier señal periódica en señales muy simples de ser analizadas: *seno* y *coseno*.
- Flexibilidad en el requisito de periodicidad: una grabación de audio es finalmente una secuencia de sonidos en un tiempo finito, de modo que repetirla infinitamente la convierte en una señal periódica.
- Pasamos de representar una señal continua en el dominio del tiempo en una señal discreta en el dominio de la frecuencia.
- Eliminar la componentes armónicas a partir de un cierto orden hace que la señal en el dominio de la frecuencia tenga una cantidad finita (y por lo tanto computable) de componentes de frecuencias.

# Observaciones

- Permite descomponer cualquier señal periódica en señales muy simples de ser analizadas: *seno* y *coseno*.
- Flexibilidad en el requisito de periodicidad: una grabación de audio es finalmente una secuencia de sonidos en un tiempo finito, de modo que repetirla infinitamente la convierte en una señal periódica.
- Pasamos de representar una señal continua en el dominio del tiempo en una señal discreta en el dominio de la frecuencia.
- Eliminar la componentes armónicas a partir de un cierto orden hace que la señal en el dominio de la frecuencia tenga una cantidad finita (y por lo tanto computable) de componentes de frecuencias.
- Esto, evidentemente, introduce un error.

# Observaciones

- Permite descomponer cualquier señal periódica en señales muy simples de ser analizadas: *seno* y *coseno*.
- Flexibilidad en el requisito de periodicidad: una grabación de audio es finalmente una secuencia de sonidos en un tiempo finito, de modo que repetirla infinitamente la convierte en una señal periódica.
- Pasamos de representar una señal continua en el dominio del tiempo en una señal discreta en el dominio de la frecuencia.
- Eliminar las componentes armónicas a partir de un cierto orden hace que la señal en el dominio de la frecuencia tenga una cantidad finita (y por lo tanto computable) de componentes de frecuencias.
- Esto, evidentemente, introduce un error.
- En la práctica, las armónicas de mayor orden suelen tener amplitudes muy pequeñas.

# Observaciones

- Permite descomponer cualquier señal periódica en señales muy simples de ser analizadas: *seno* y *coseno*.
- Flexibilidad en el requisito de periodicidad: una grabación de audio es finalmente una secuencia de sonidos en un tiempo finito, de modo que repetirla infinitamente la convierte en una señal periódica.
- Pasamos de representar una señal continua en el dominio del tiempo en una señal discreta en el dominio de la frecuencia.
- Eliminar las componentes armónicas a partir de un cierto orden hace que la señal en el dominio de la frecuencia tenga una cantidad finita (y por lo tanto computable) de componentes de frecuencias.
- Esto, evidentemente, introduce un error.
- En la práctica, las armónicas de mayor orden suelen tener amplitudes muy pequeñas.
- Por lo tanto, el error introducido por la frecuencia de corte no es apreciable en la práctica.

## 1 Fundamentos

## 2 Procesamiento de Señales digitales

- Digitalización de la señal
- Arquitecturas de Procesamiento de una señal digital

## 3 Modelo de ejecución SIMD

## 4 Implementaciones SIMD en x86

## 5 Instrucciones

## 6 Advanced Vector Extensions

# Señal digitalizada

- El proceso de digitalización de una señal responde al siguiente modelo.

# Señal digitalizada

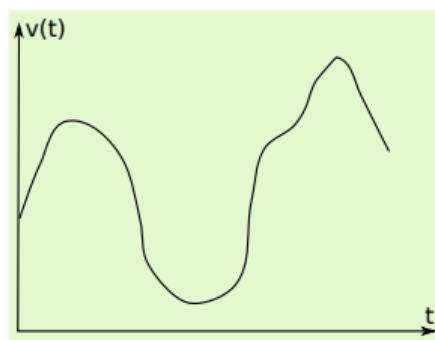
- El proceso de digitalización de una señal responde al siguiente modelo.



- La función  $v(t)$ , es una tensión de entrada analógica.

# Señal digitalizada

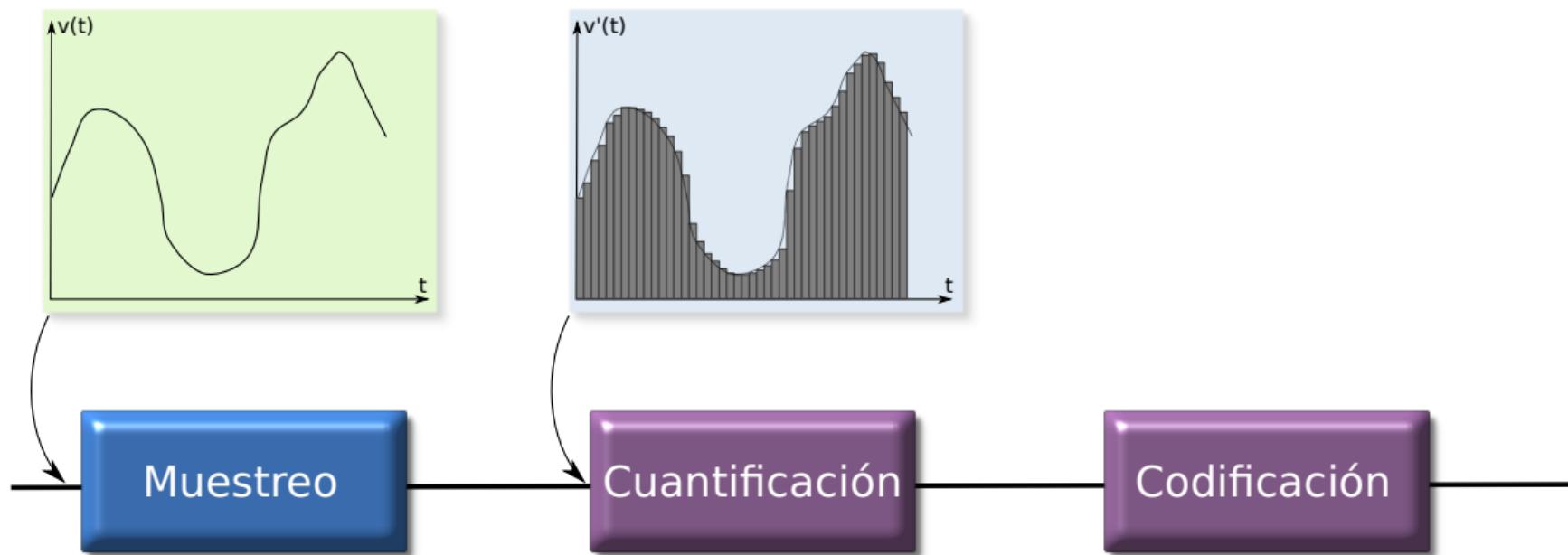
- El proceso de digitalización de una señal responde al siguiente modelo.



- La función  $v(t)$ , es una tensión de entrada analógica.

# Señal digitalizada

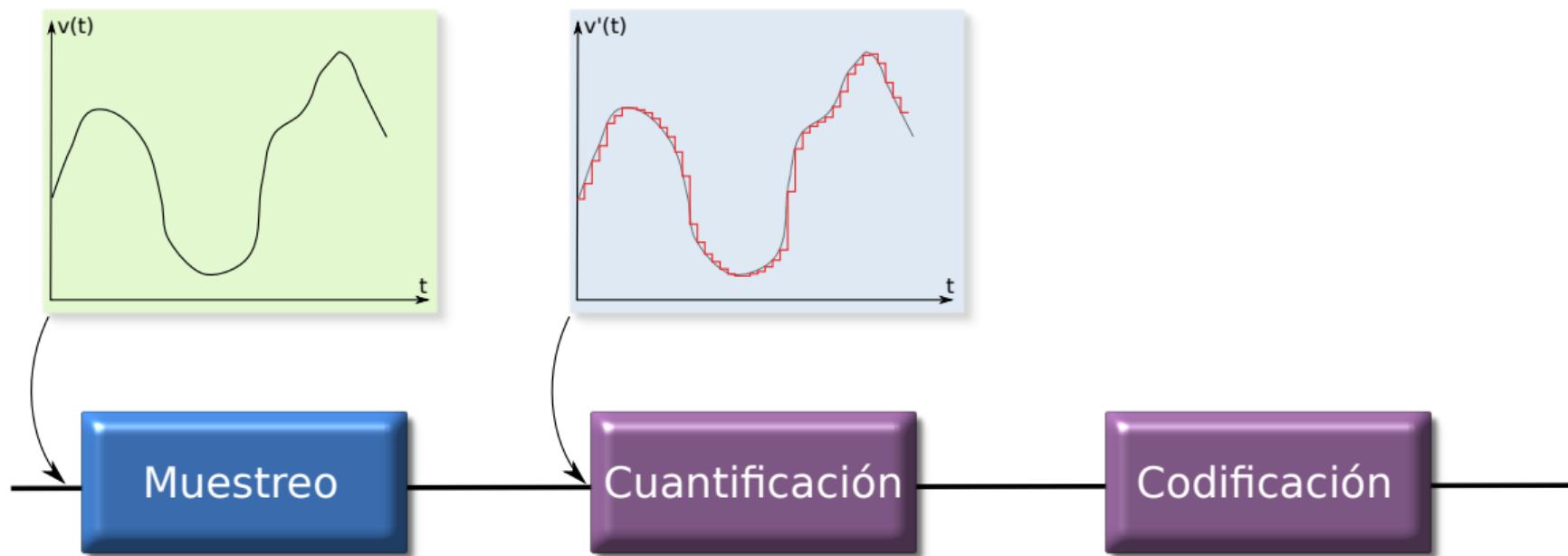
- El proceso de digitalización de una señal responde al siguiente modelo.



- La función  $v'(t)$ , es una función discreta, que solo cambia entre valores discretos (no continua) a intervalos regulares, dados por la frecuencia de muestreo, y mantiene su valor durante el intervalo.

# Señal digitalizada

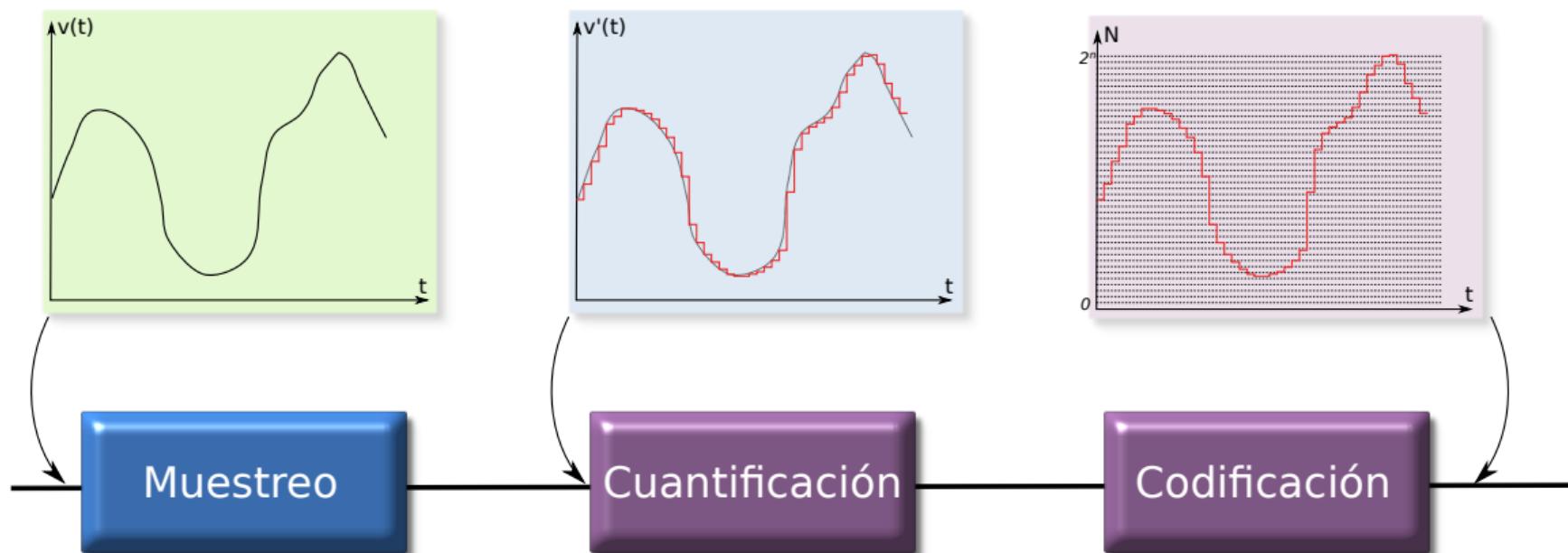
- El proceso de digitalización de una señal responde al siguiente modelo.



- La función  $v'(t)$ , es una función discreta, que solo cambia entre valores discretos (no continua) a intervalos regulares, dados por la frecuencia de muestreo, y mantiene su valor durante el intervalo.

# Señal digitalizada

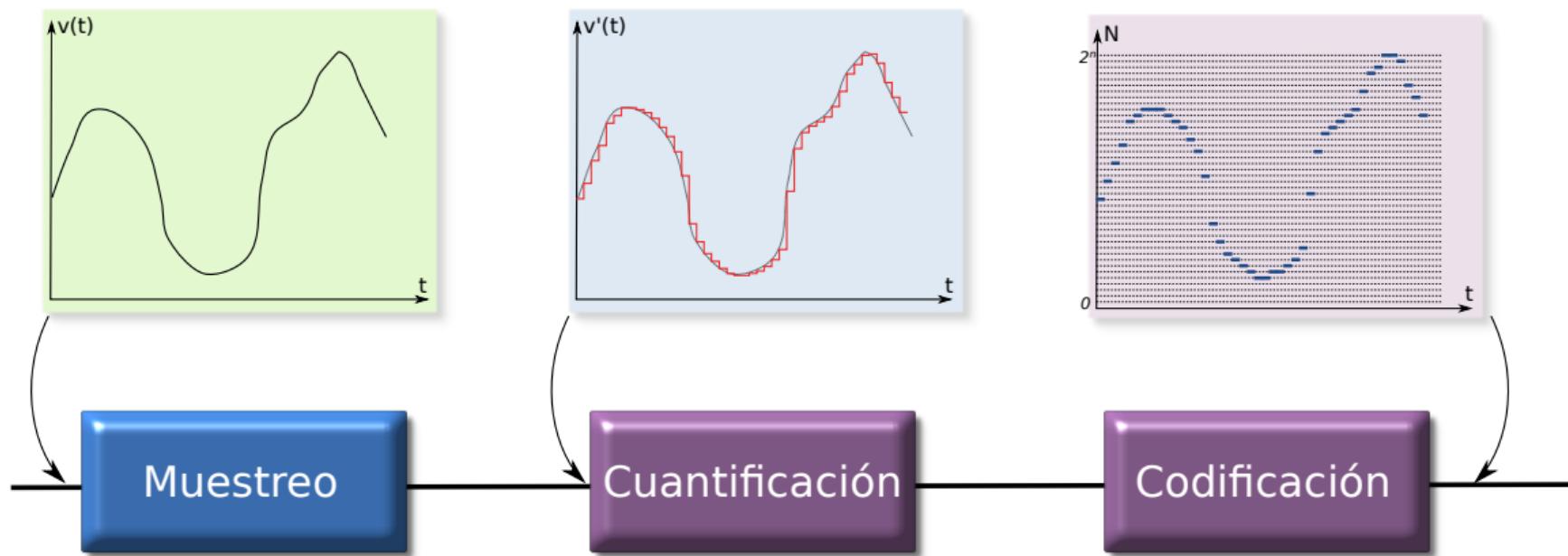
- El proceso de digitalización de una señal responde al siguiente modelo.



- El proceso de Cuantificación (o Cuantización), transforma los valores obtenidos de la variable dependiente aproximándolos al valor mas cercano de un conjunto finito de  $2^n$  números.

# Señal digitalizada

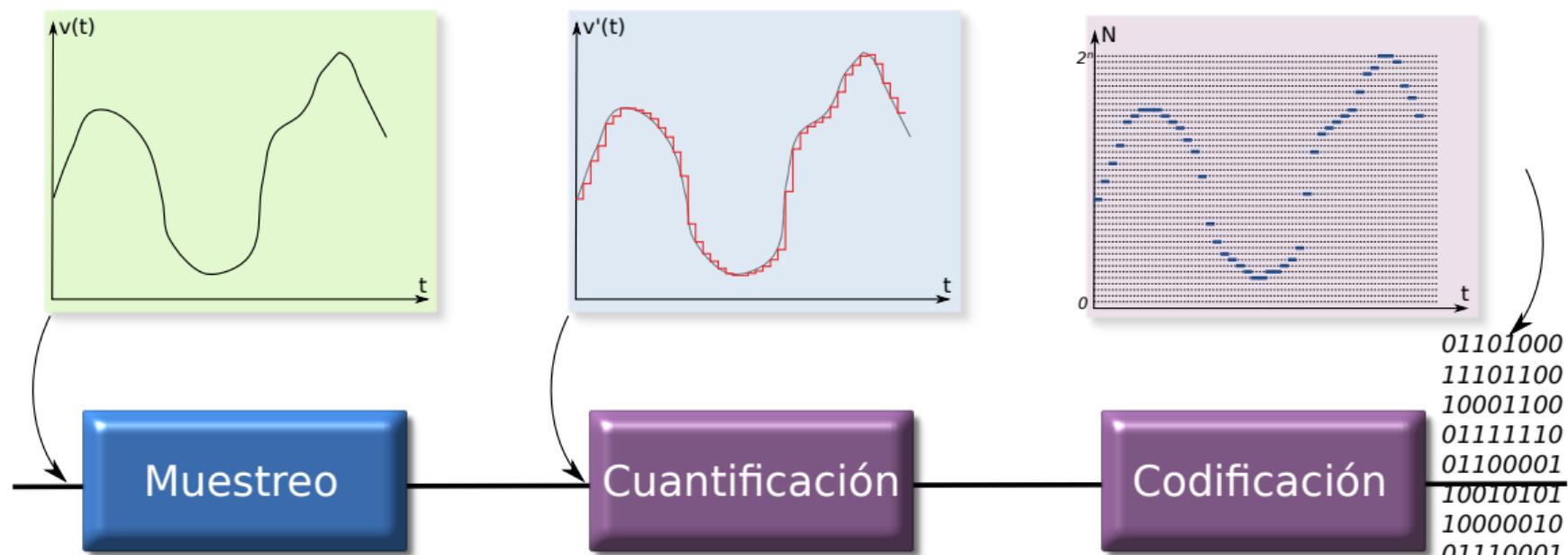
- El proceso de digitalización de una señal responde al siguiente modelo.



- El proceso de Cuantificación (o Cuantización), transforma los valores obtenidos de la variable dependiente aproximándolos al valor mas cercano de un conjunto finito de  $2^n$  números.

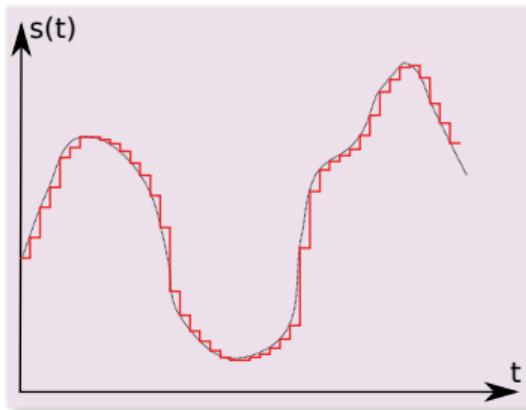
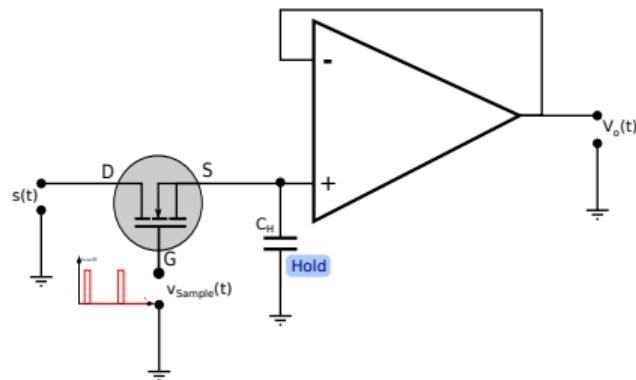
# Señal digitalizada

- El proceso de digitalización de una señal responde al siguiente modelo.



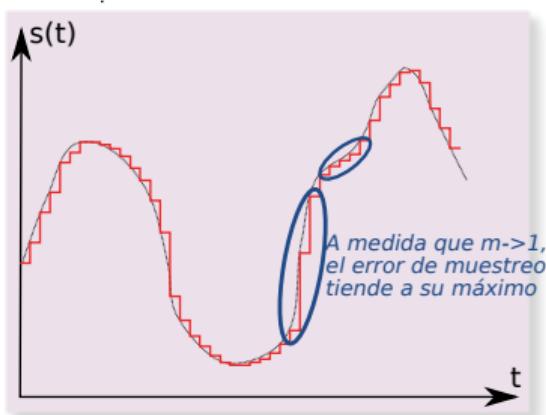
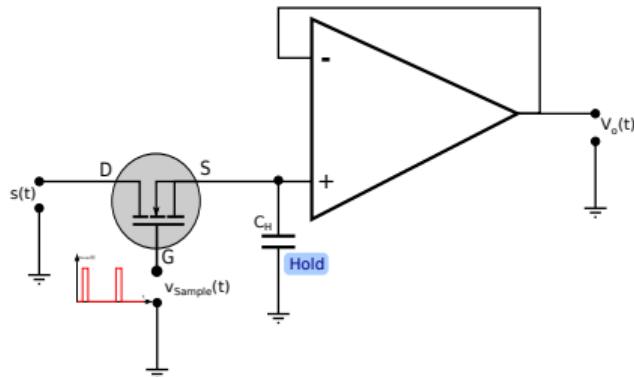
- El proceso de Cuantificación (o Cuantización), transforma los valores obtenidos de la variable dependiente aproximándolos al valor mas cercano de un conjunto finito de  $2^n$  números.

# 1er. Fase: Muestreo y retención



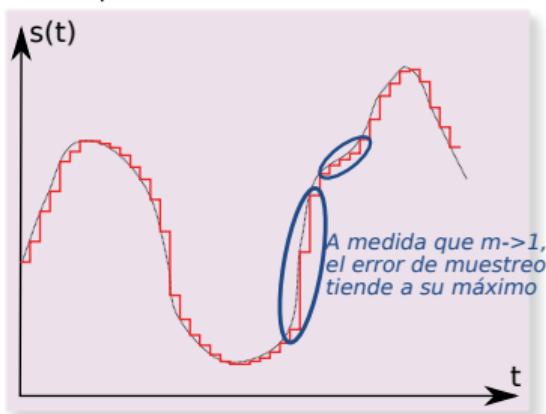
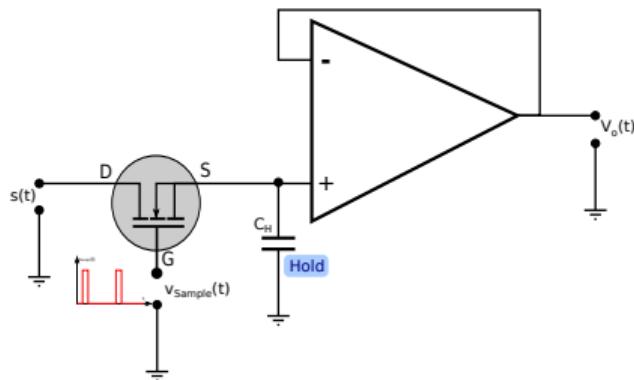
- Se toma un valor instantáneo de la señal y se “retiene” el valor de tensión en una capacidad, con un circuito resistivo de descarga de muy alta resistencia.

# 1er. Fase: Muestreo y retención



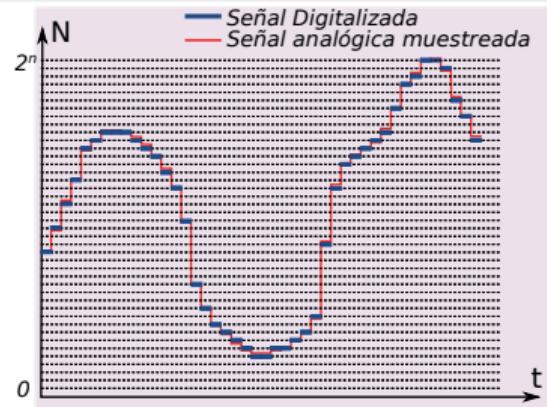
- Se toma un valor instantáneo de la señal y se “retiene” el valor de tensión en una capacidad, con un circuito resistivo de descarga de muy alta resistencia.
- Comparando la señal saliente del circuito de Muestreo y Retención, con la señal original (en punteado suave), observar que el error aumenta en los cambios abruptos de señal.*

# 1er. Fase: Muestreo y retención



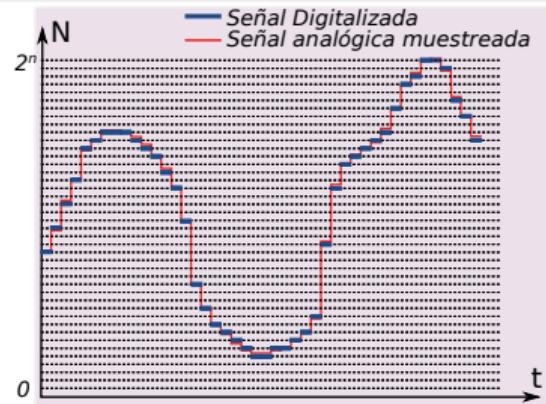
- Se toma un valor instantáneo de la señal y se “retiene” el valor de tensión en una capacidad, con un circuito resistivo de descarga de muy alta resistencia.
- Comparando la señal saliente del circuito de Muestreo y Retención, con la señal original (en punteado suave), observar que el error aumenta en los cambios abruptos de señal.*
- Otra forma de observar la relación entre la frecuencia de muestreo y la máxima frecuencia de la señal original

# Cuantificación y Codificación



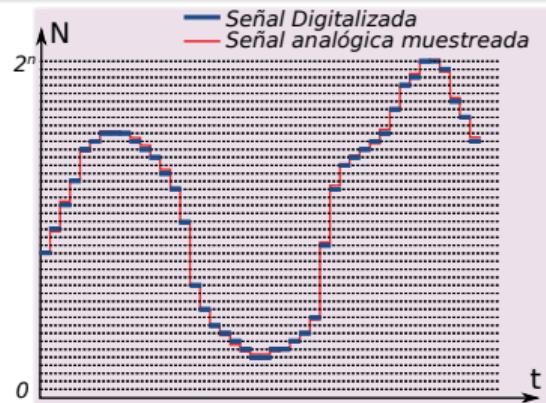
- Se llevan a cabo en un Conversor Analógico Digital.

# Cuantificación y Codificación



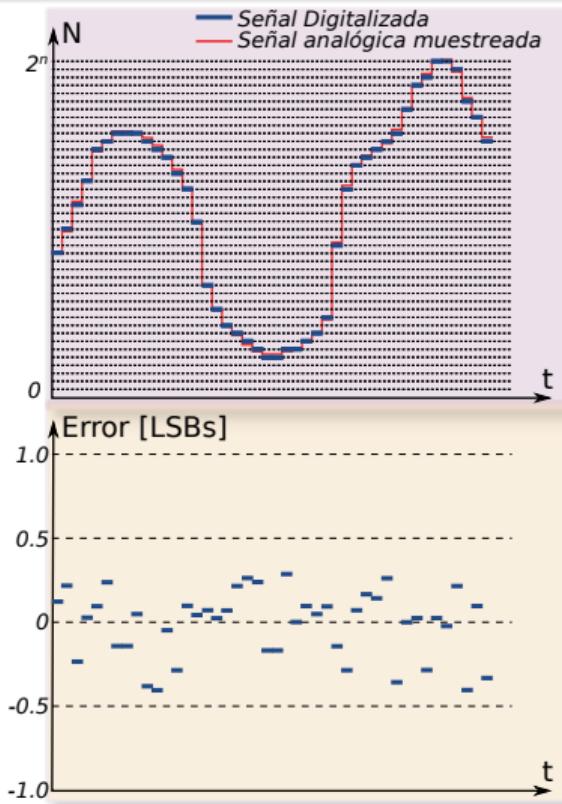
- Se llevan a cabo en un Conversor Analógico Digital.
- Se puede aproximar por redondeo o truncamiento

# Cuantificación y Codificación



- Se llevan a cabo en un Conversor Analógico Digital.
- Se puede aproximar por redondeo o truncamiento
- Independientemente del método el error aumenta en las zonas donde la derivada de la señal es mas alta.

# Cuantificación y Codificación



- Se llevan a cabo en un Conversor Analógico Digital.
- Se puede aproximar por redondeo o truncamiento
- Independientemente del método el error aumenta en las zonas donde la derivada de la señal es mas alta.
- Si tomamos la diferencia entre la señal muestreada y la cuantificada y operamos una diferencia se obtiene el error de Cuantificación o Cuan-

## 1 Fundamentos

## 2 Procesamiento de Señales digitales

- Digitalización de la señal
- Arquitecturas de Procesamiento de una señal digital

## 3 Modelo de ejecución SIMD

## 4 Implementaciones SIMD en x86

## 5 Instrucciones

## 6 Advanced Vector Extensions

# Procesador de Señales Digitales

- Es una CPU de propósito dedicado, diseñada para realizar cálculos y procesamiento de un único tipo de datos: secuencias de valores correspondientes a la codificación de las muestras de una señal de entrada.
- Su arquitectura está pensada para optimizar el procesamiento de datos que no son de gran tamaño (8, 16, 24, o a lo sumo 32 bits). Se trata de píxeles de una imagen, o de valores instantáneos de audio, o de señales médicas o de mapas térmicos, etc.
- La característica distintiva de este tipo de datos reside en sus algoritmos de cálculo: Normalmente se requiere procesar no solo el valor actual sino la combinación del valor actual con  $n$  valores anteriores en el tiempo, o vecinos (en el caso de una imagen lo que llamaremos  $N_8$ )

# Procesador de Señales Digitales

- En general una operación muy frecuente son los filtros de convolución, dados por una expresión del tipo:

$$y[n] = \sum_{i=0}^N a_i \cdot x[n-i] = a_0 \cdot x[n] + a_1 \cdot x[n-1] + a_2 \cdot x[n-2] + \dots + a_N \cdot x[n-N]$$

- En general se deben resolver sumas de productos y acumular su resultado.
- Para optimizar se deberían leer y procesar varios datos en paralelo
- Las técnicas de paralelismo que se desarrollaron en estos procesadores se denominaron por tal razón **Data Level Parallelism**.

# Arquitectura de un Procesador de Señales Digitales

- En general el diseño de un Procesador de Señales Digitales concentra sus esfuerzos en resolver en paralelo:

# Arquitectura de un Procesador de Señales Digitales

- En general el diseño de un Procesador de Señales Digitales concentra sus esfuerzos en resolver en paralelo:

**El acceso a los operandos** Para este fin se implementan buses paralelos con una cantidad de líneas de datos superior al ancho de palabra de la CPU

# Arquitectura de un Procesador de Señales Digitales

- En general el diseño de un Procesador de Señales Digitales concentra sus esfuerzos en resolver en paralelo:

**El acceso a los operandos** Para este fin se implementan buses paralelos con una cantidad de líneas de datos superior al ancho de palabra de la CPU

**El almacenamiento de resultados** Se resuelve mediante buses dedicados para manejar la salida de la ALU hacia memoria o registros (con las consideraciones que la concurrencia de accesos debe tener en cuenta en el hardware)

# Arquitectura de un Procesador de Señales Digitales

- En general el diseño de un Procesador de Señales Digitales concentra sus esfuerzos en resolver en paralelo:

**El acceso a los operandos** Para este fin se implementan buses paralelos con una cantidad de líneas de datos superior al ancho de palabra de la CPU

**El almacenamiento de resultados** Se resuelve mediante buses dedicados para manejar la salida de la ALU hacia memoria o registros (con las consideraciones que la concurrencia de accesos debe tener en cuenta en el hardware)

**El procesamiento de la mayor cantidad de datos posible** Los primeros pasos se conocen como VLIW (Very Large Instruction Word), que derivó en el modelo de ejecución SIMD (Single Instruction Multiple Data)

1 Fundamentos

2 Procesamiento de Señales digitales

3 Modelo de ejecución SIMD

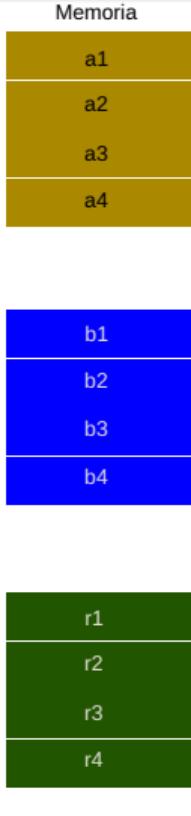
- Un modelo de paralelización

4 Implementaciones SIMD en x86

5 Instrucciones

6 Advanced Vector Extensions

# Single Instruction Multiple Data



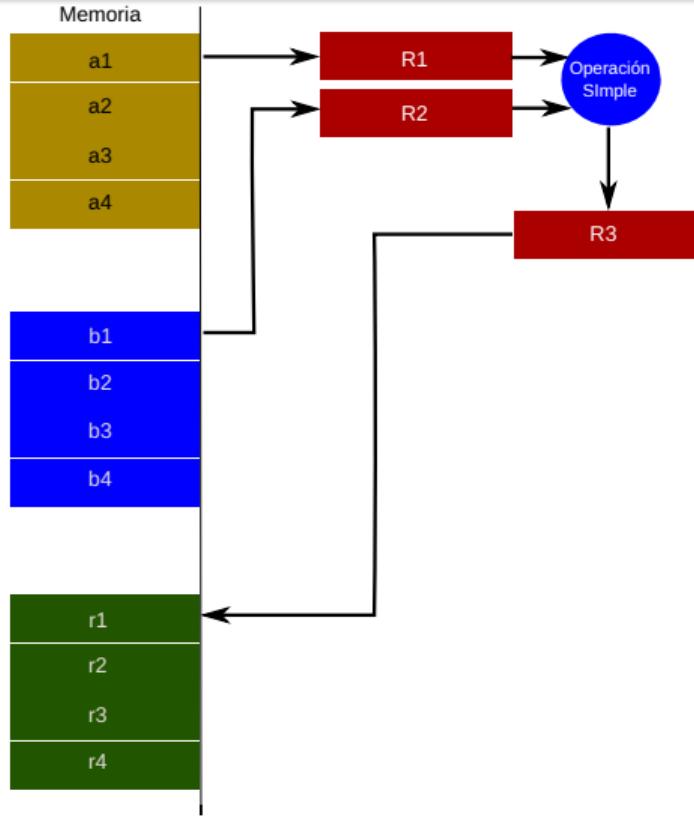
- Se trata de un modelo de ejecución capaz de computar una sola operación sobre un conjunto de múltiples datos.
- Se refiere a esta técnica como paralelismo a nivel de datos.
- Es particularmente útil para procesar audio, video, o imágenes en donde se aplican algoritmos repetitivos sobre sets de datos del mismo formato y que se procesan en conjunto, como por ejemplo en filtros, compresores, codificadores en donde la salida depende de los últimos  $n$  valores de muestras tomados.
- La figura muestra el layout típico de variables memoria para aplicar este modelo

# Como sería la vida sin SIMD?

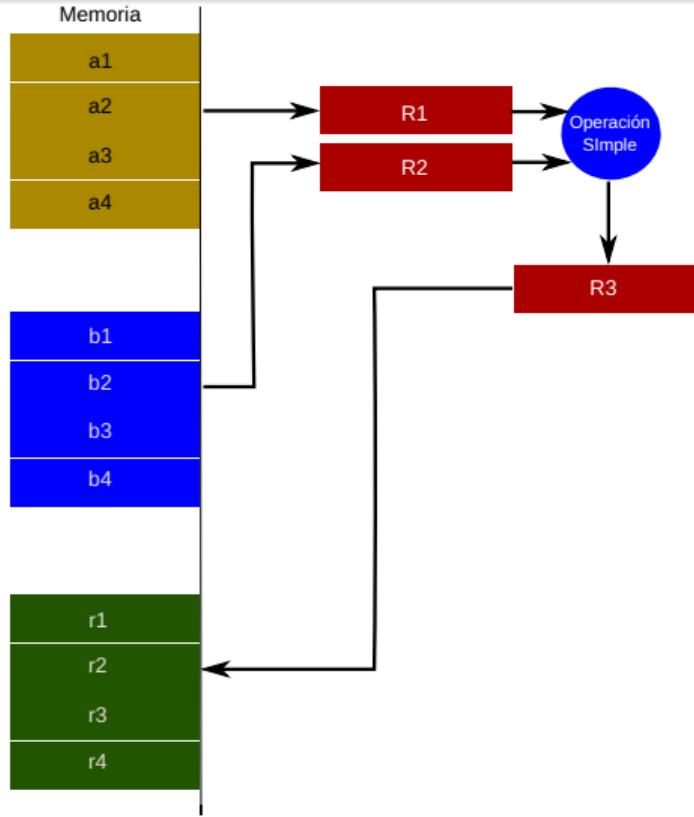
Memoria
a1
a2
a3
a4
b1
b2
b3
b4
r1
r2
r3
r4

- En contraposición a SIMD el modelo previo es SISD (**S**ingle **I**nstruction **S**ingle **D**ata)
- Considerando el sector de memoria de la figura, nos proponemos realizar una operación aritmética o lógica sobre las cadenas de datos  $a_n$  y  $b_n$ , almacenando el resultado en  $r_n$
- Si un procesador no dispone de un modelo arquitectural que le permita implementar paralelismo a nivel de datos, como SIMD, esta operación implica un loop, ya que solo puede procesar un dato por cada instrucción (SISD).

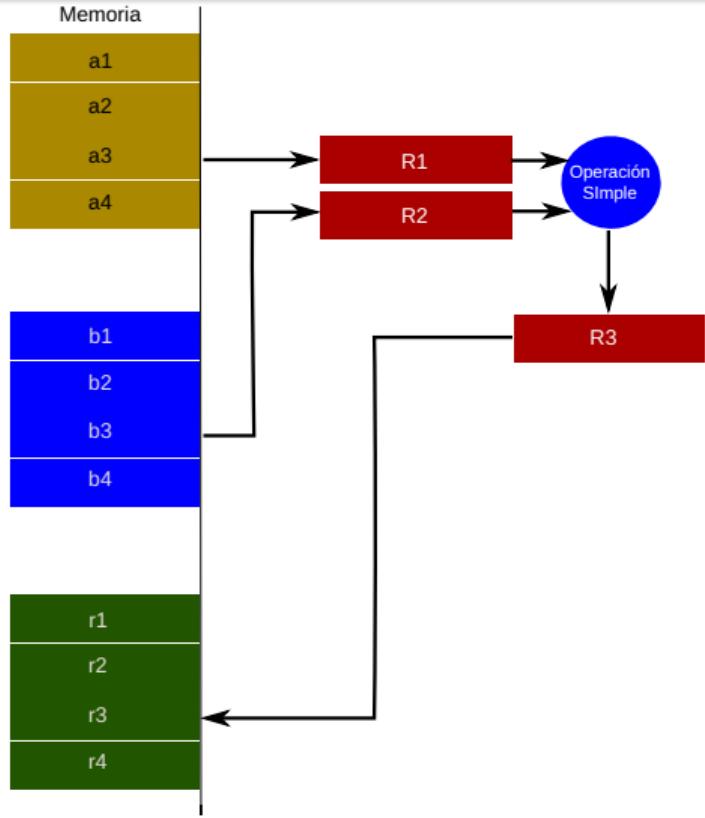
# Single Instruction Single Data



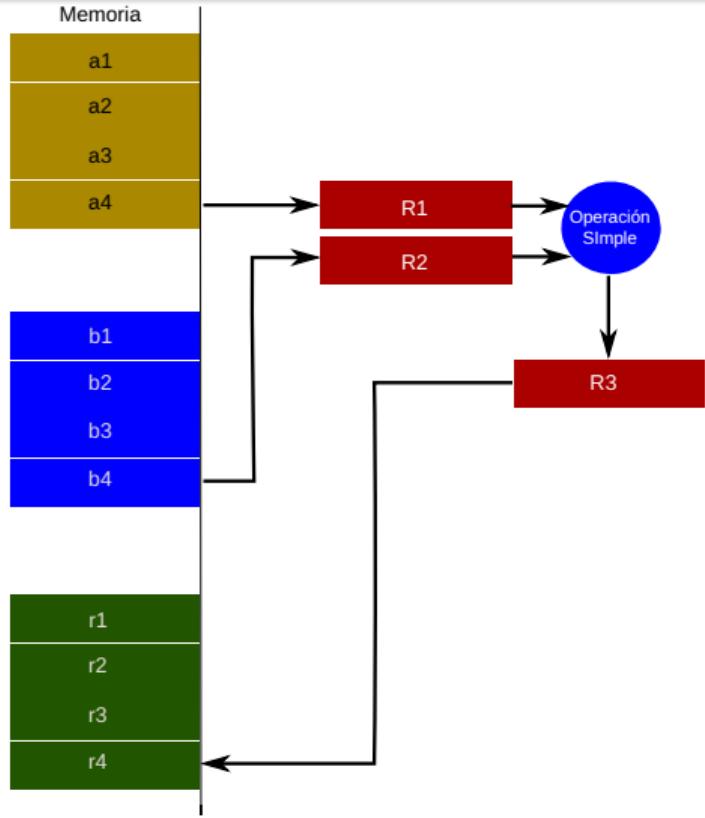
# Single Instruction Single Data



# Single Instruction Single Data

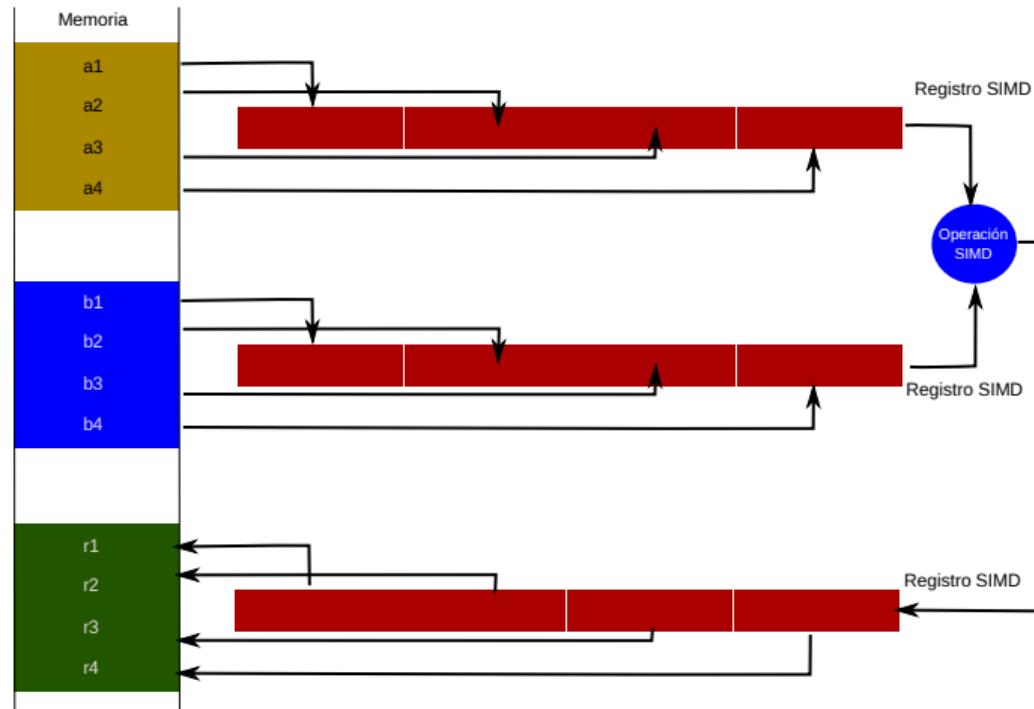


# Single Instruction Single Data

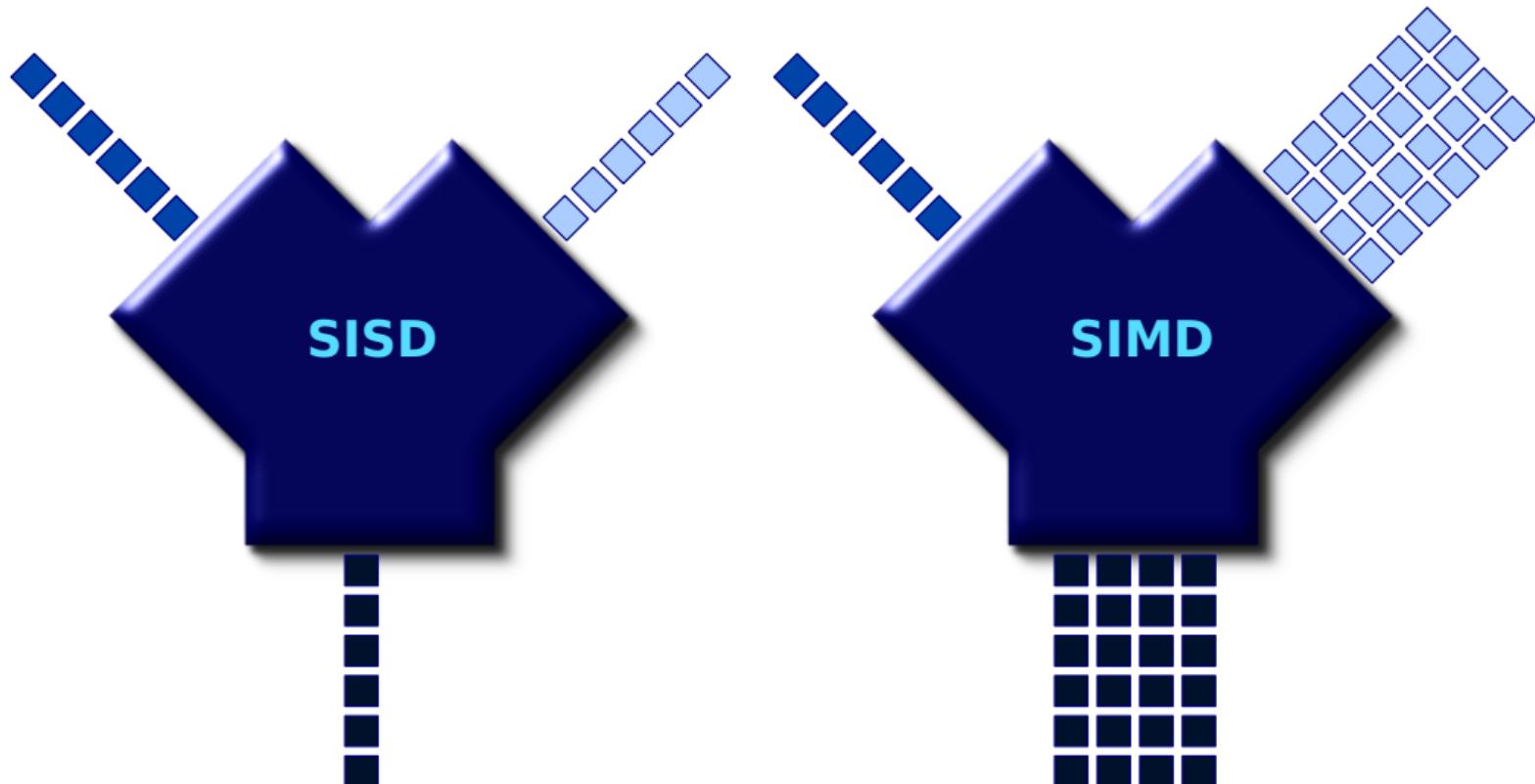


# Single Instruction Multiple Data

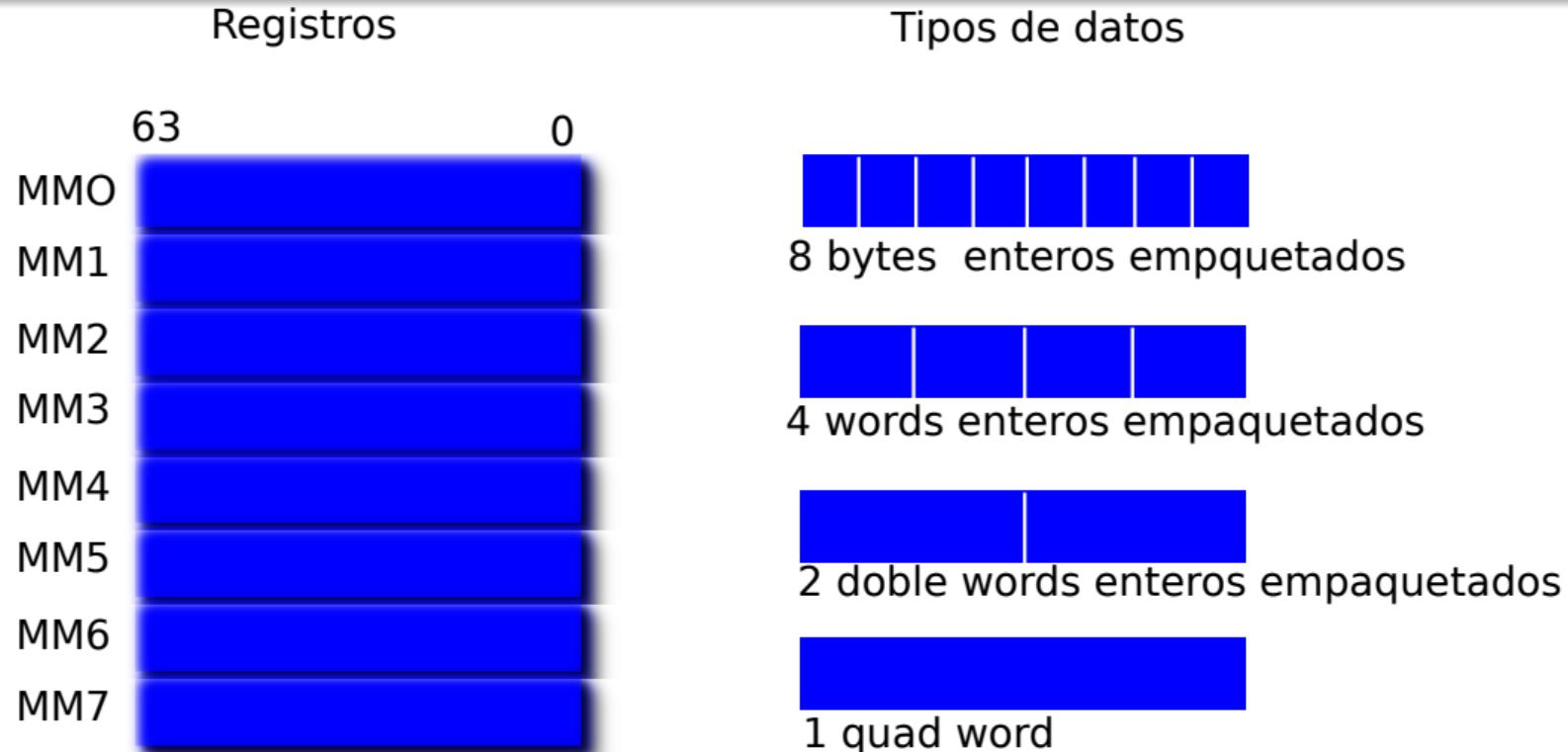
- Cada Registro se carga en una sola instrucción
- La operación se efectúa en la segunda instrucción



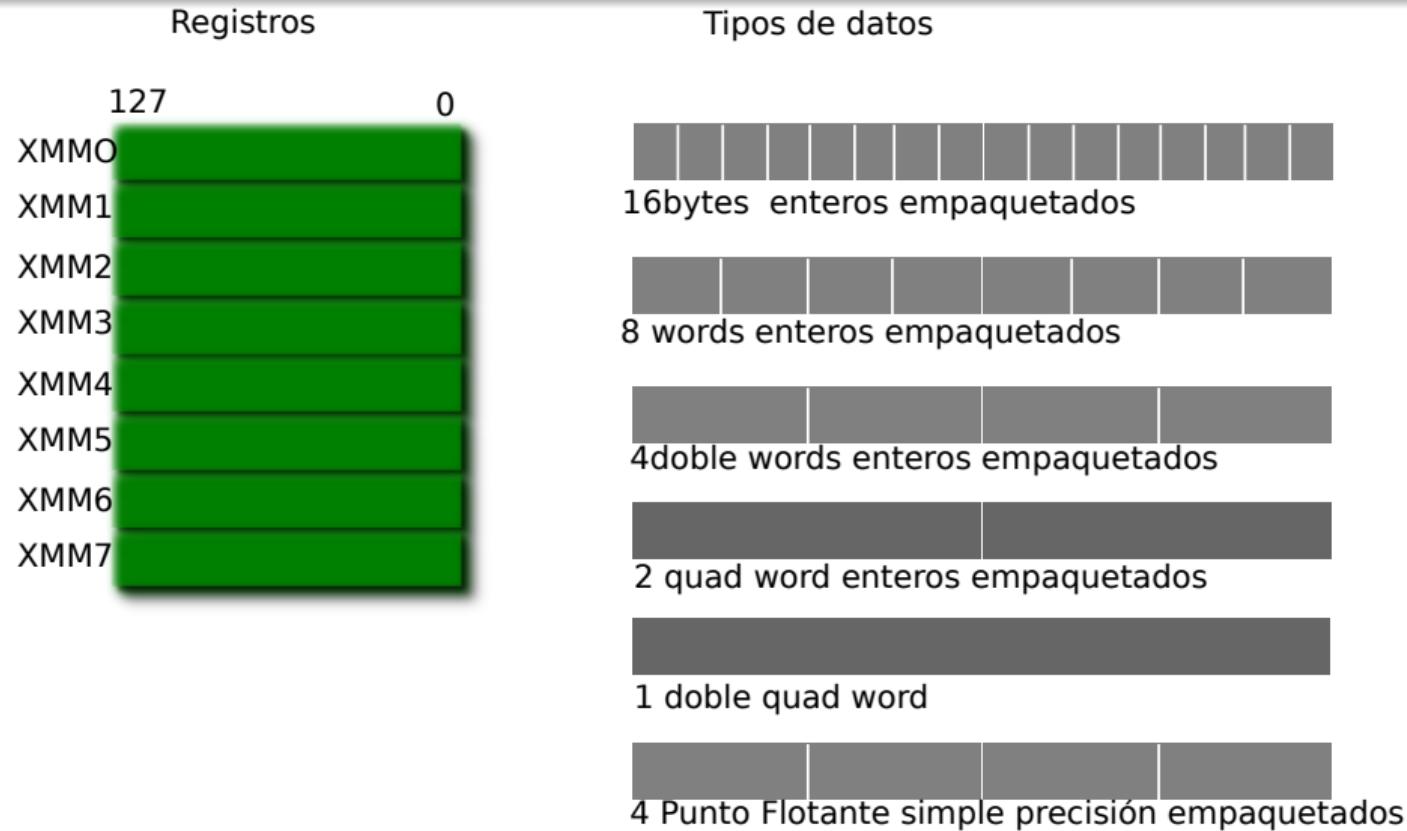
# Resumiendo, SIMD vs. SISD



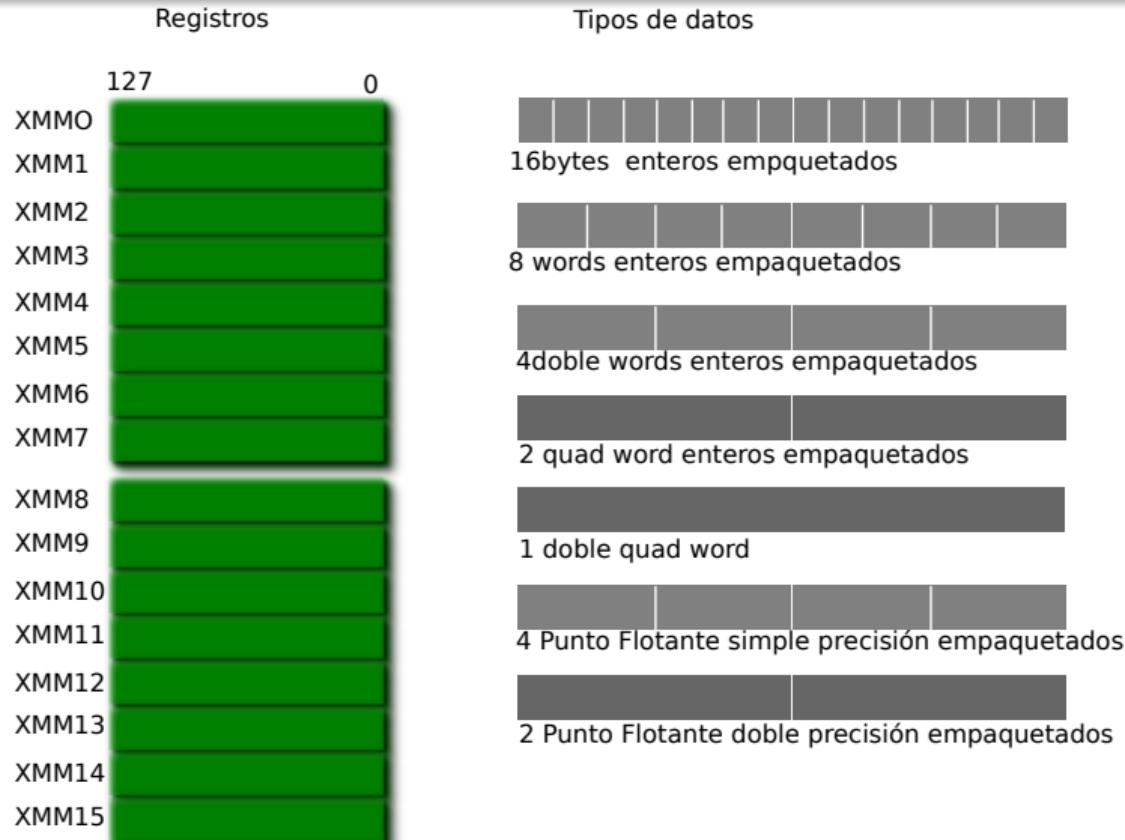
# Multimedia extensions - MMX



# Streaming SIMD Extension



# Streaming SIMD Extension en Modo 64 bits



1 Fundamentos

2 Procesamiento de Señales digitales

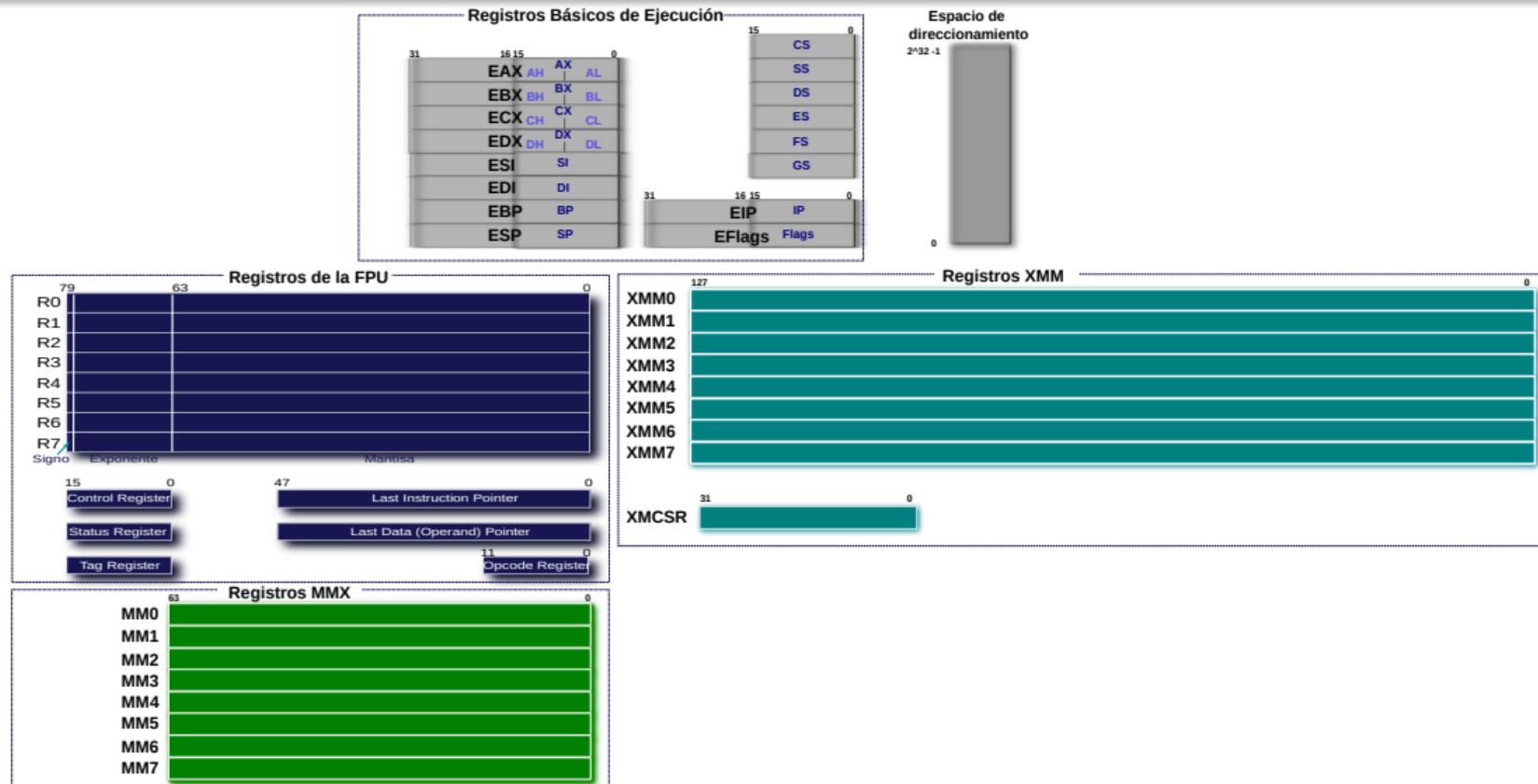
3 Modelo de ejecución SIMD

## 4 Implementaciones SIMD en x86

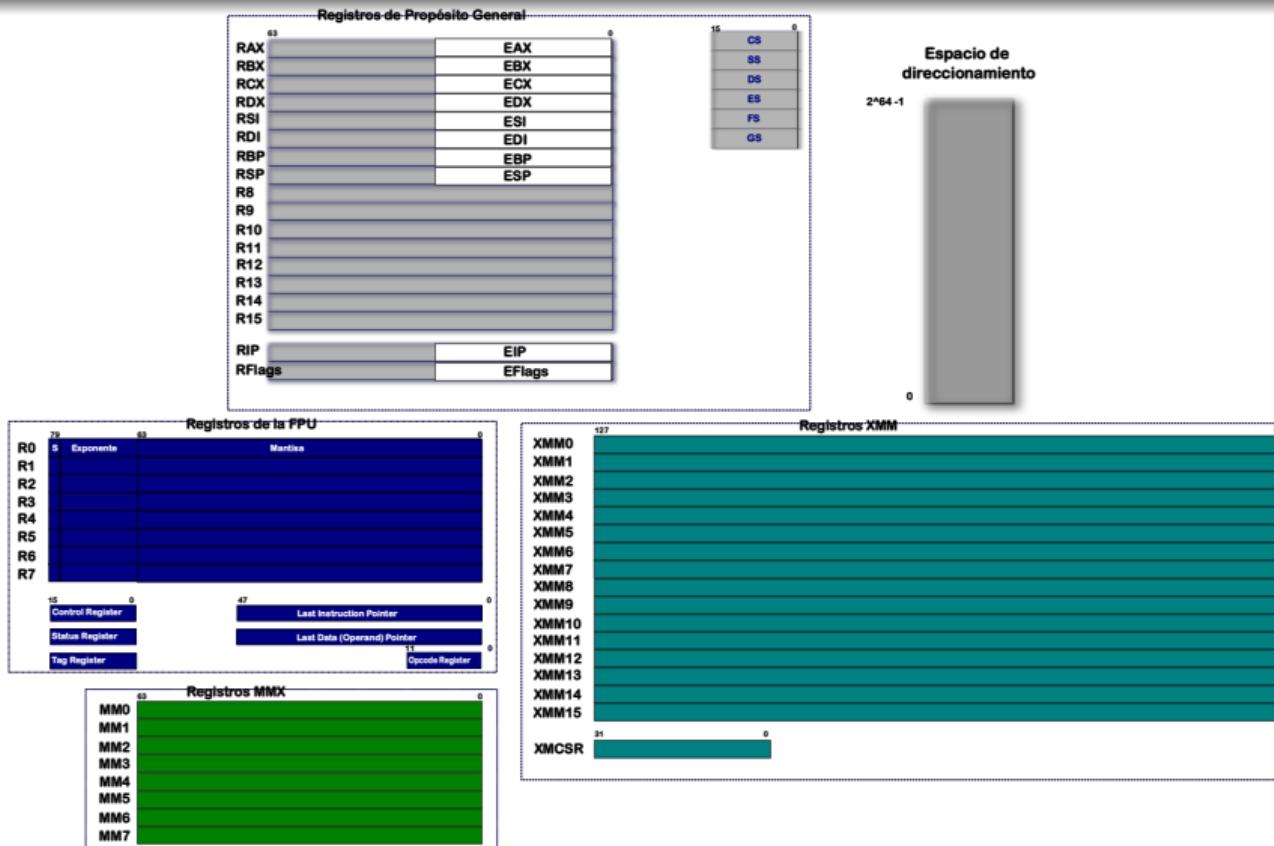
- Arquitectura completa
- Números Reales
- Formatos de Punto Fijo
- Formatos de Punto Flotante
- Codificación de Números Reales

5 Instrucciones

## ISA en 32 bits



## ISA en 64 bits



1 Fundamentos

2 Procesamiento de Señales digitales

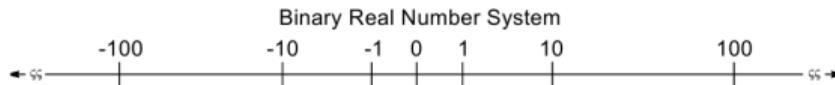
3 Modelo de ejecución SIMD

## 4 Implementaciones SIMD en x86

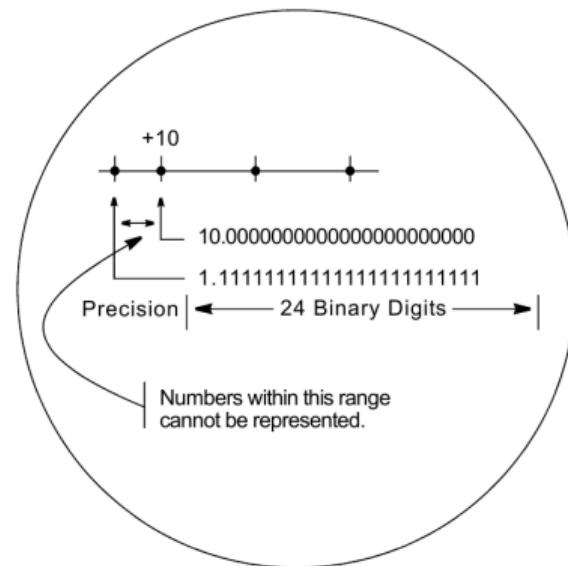
- Arquitectura completa
- Números Reales
- Formatos de Punto Fijo
- Formatos de Punto Flotante
- Codificación de Números Reales

5 Instrucciones

# Números Reales

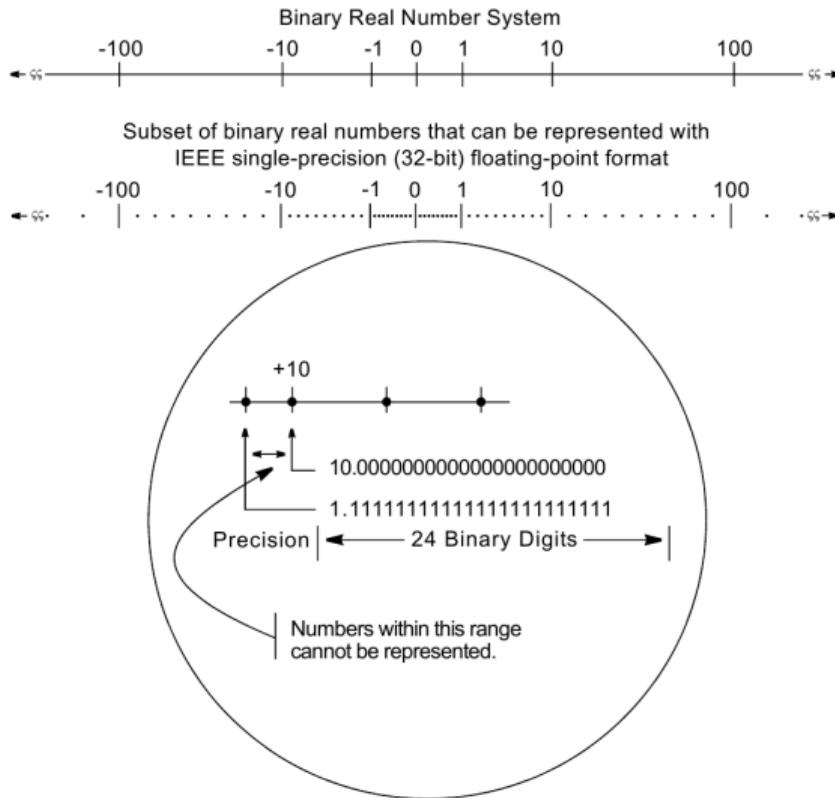


Subset of binary real numbers that can be represented with IEEE single-precision (32-bit) floating-point format



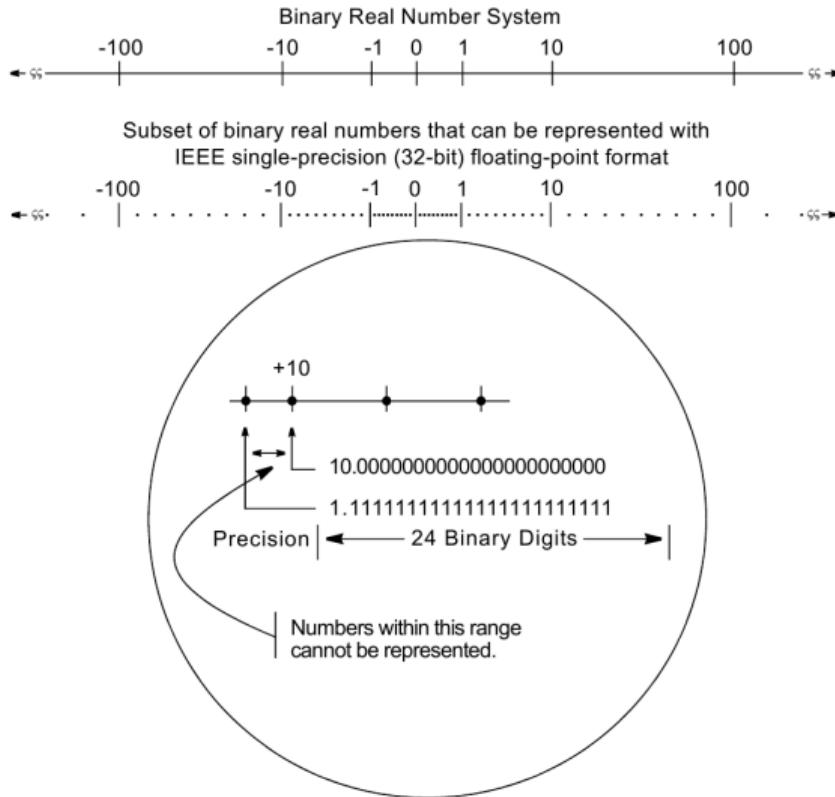
- El rango de los números reales comprende desde  $-\infty$  hasta  $+\infty$ .

# Números Reales



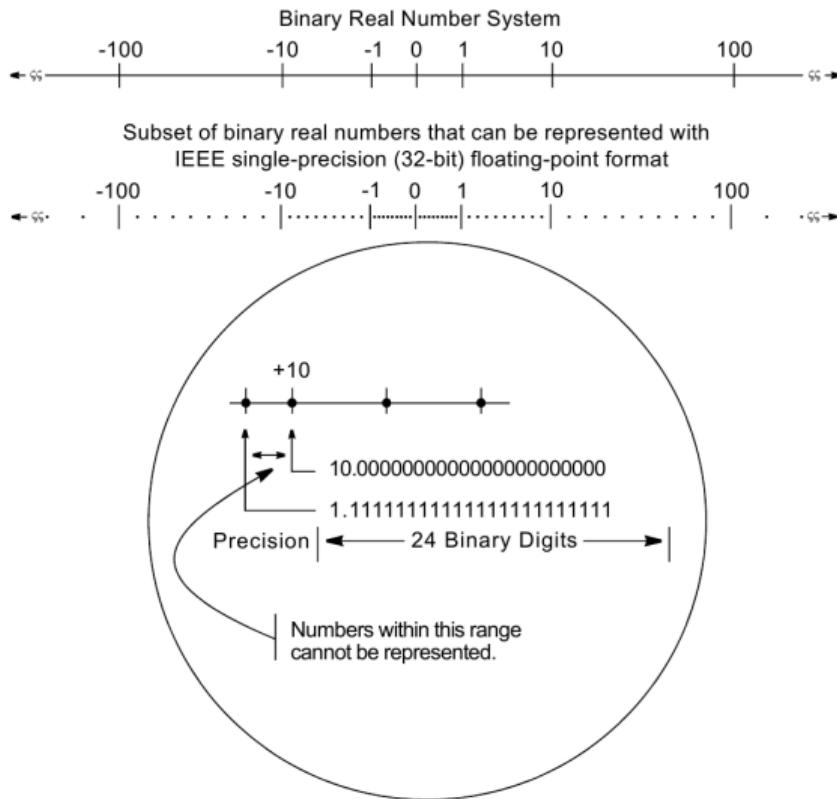
- El rango de los números reales comprende desde  $-\infty$  hasta  $+\infty$ .
- Los registros de un procesador tienen resolución finita.

# Números Reales



- El rango de los números reales comprende desde  $-\infty$  hasta  $+\infty$ .
- Los registros de un procesador tienen resolución finita.
- Por lo tanto un computador solo puede representar un subconjunto de  $\mathbb{R}$ .

# Números Reales



- El rango de los números reales comprende desde  $-\infty$  hasta  $+\infty$ .
- Los registros de un procesador tienen resolución finita.
- Por lo tanto un computador solo puede representar un subconjunto de  $\mathbb{R}$ .
- Además, no es solo un tema de magnitud sino de resolución.

# Representación binaria de Números Reales

## Formatos

En general podemos formalizar la representación de un número real expresado en los siguientes formatos:

- ① Punto Fijo
- ② Punto Flotante

# Notación Científica

- Para el caso de los números reales se trabaja en **notación científica**.

$$n = \pm f * 10^e$$

# Notación Científica

- Para el caso de los números reales se trabaja en **notación científica**.

$$n = \pm f * 10^e$$

$$-725,832 = -7,25832 * 10^2 = -725,832 * 10^0$$

# Notación Científica

- Para el caso de los números reales se trabaja en **notación científica**.

$$n = \pm f * 10^e$$

$$-725,832 = -7,25832 * 10^2 = -725,832 * 10^0$$

$$3,14 = 0,314 * 10^1 = 3,14 * 10^0$$

# Notación Científica

- Para el caso de los números reales se trabaja en **notación científica**.

$$n = \pm f * 10^e$$

$$-725,832 = -7,25832 * 10^2 = -725,832 * 10^0$$

$$3,14 = 0,314 * 10^1 = 3,14 * 10^0$$

$$0,000001 = 0,1 * 10^{-5} = 1,0 * 10^{-6}$$

# Notación Científica

- Para el caso de los números reales se trabaja en **notación científica**.

$$n = \pm f * 10^e$$

$$-725,832 = -7,25832 * 10^2 = -725,832 * 10^0$$

$$3,14 = 0,314 * 10^1 = 3,14 * 10^0$$

$$0,000001 = 0,1 * 10^{-5} = 1,0 * 10^{-6}$$

$$1941 = 0,1941 * 10^4 = 1,941 * 10^3$$

# Notación Científica

- Para el caso de los números reales se trabaja en **notación científica**.

$$n = \pm f * 10^e$$

$$-725,832 = -7,25832 * 10^2 = -725,832 * 10^0$$

$$3,14 = 0,314 * 10^1 = 3,14 * 10^0$$

$$0,000001 = 0,1 * 10^{-5} = 1,0 * 10^{-6}$$

$$1941 = 0,1941 * 10^4 = 1,941 * 10^3$$

- Para unificar la representación se recurre a la **notación científica normalizada**, en donde:

$0,1 \leq f < 1$ , y  $e$  es un entero con signo.

# Notación Científica en el sistema Binario

- En el sistema binario la expresión de un número en notación científica normalizada es:

$$n = \pm f * 2^e$$

- En donde:

$0,5 \leq f < 1$ , y  $e$  es un entero con signo.

1 Fundamentos

2 Procesamiento de Señales digitales

3 Modelo de ejecución SIMD

## 4 Implementaciones SIMD en x86

- Arquitectura completa
- Números Reales
- **Formatos de Punto Fijo**
- Formatos de Punto Flotante
- Codificación de Números Reales

5 Instrucciones

# Representación binaria en Punto Fijo con signo

- Se representan mediante una expresión del tipo:

$$(a_n a_{n-1} \dots a_0.a_{-1} a_{-2} \dots a_{-m})_2 = (-1)^s * (a_n * 2^n + \dots + a_0 * 2^0 + a_{-1} * 2^{-1} + a_{-2} * 2^{-2} + \dots + a_{-m} * 2^{-m})$$

- Donde:

- $s$  es el signo: **0** si el número es positivo y **1** si es negativo
  - $a_i \in \mathbb{Z}$  y  $0 \leq a_i \leq 1, \forall i = -m, -1, 0, 1, \dots, n$

- Distancia entre dos números consecutivos es  $2^{-m}$ .
- *Deja de ser un rango continuo de números para transformarse en un rango discreto.*

1 Fundamentos

2 Procesamiento de Señales digitales

3 Modelo de ejecución SIMD

## 4 Implementaciones SIMD en x86

- Arquitectura completa
- Números Reales
- Formatos de Punto Fijo
- Formatos de Punto Flotante**
- Codificación de Números Reales

5 Instrucciones

# Representación en Punto Flotante

- Se representan con los pares de valores  $(m, e)$ , denotando:

$$(m, e) = m * b^e$$

- En donde:

- ① **m** llamado mantisa, y que representa un número fraccionario.
- ② **e** llamado exponente, al cual se debe elevar la base numérica (**b**) de representación para obtener el valor real.

# Representación en Punto Flotante

# Representación en Punto Flotante

- Mantisa y exponente pueden representarse:

# Representación en Punto Flotante

- Mantisa y exponente pueden representarse:
  - ➊ con signo.

# Representación en Punto Flotante

- Mantisa y exponente pueden representarse:
  - 1 con signo.
  - 2 sin signo.

# Representación en Punto Flotante

- Mantisa y exponente pueden representarse:
  - 1 con signo.
  - 2 sin signo.
  - 3 con notación complemento.

# Representación en Punto Flotante

- Mantisa y exponente pueden representarse:

- ① con signo.
- ② sin signo.
- ③ con notación complemento.
- ④ con notación exceso m.

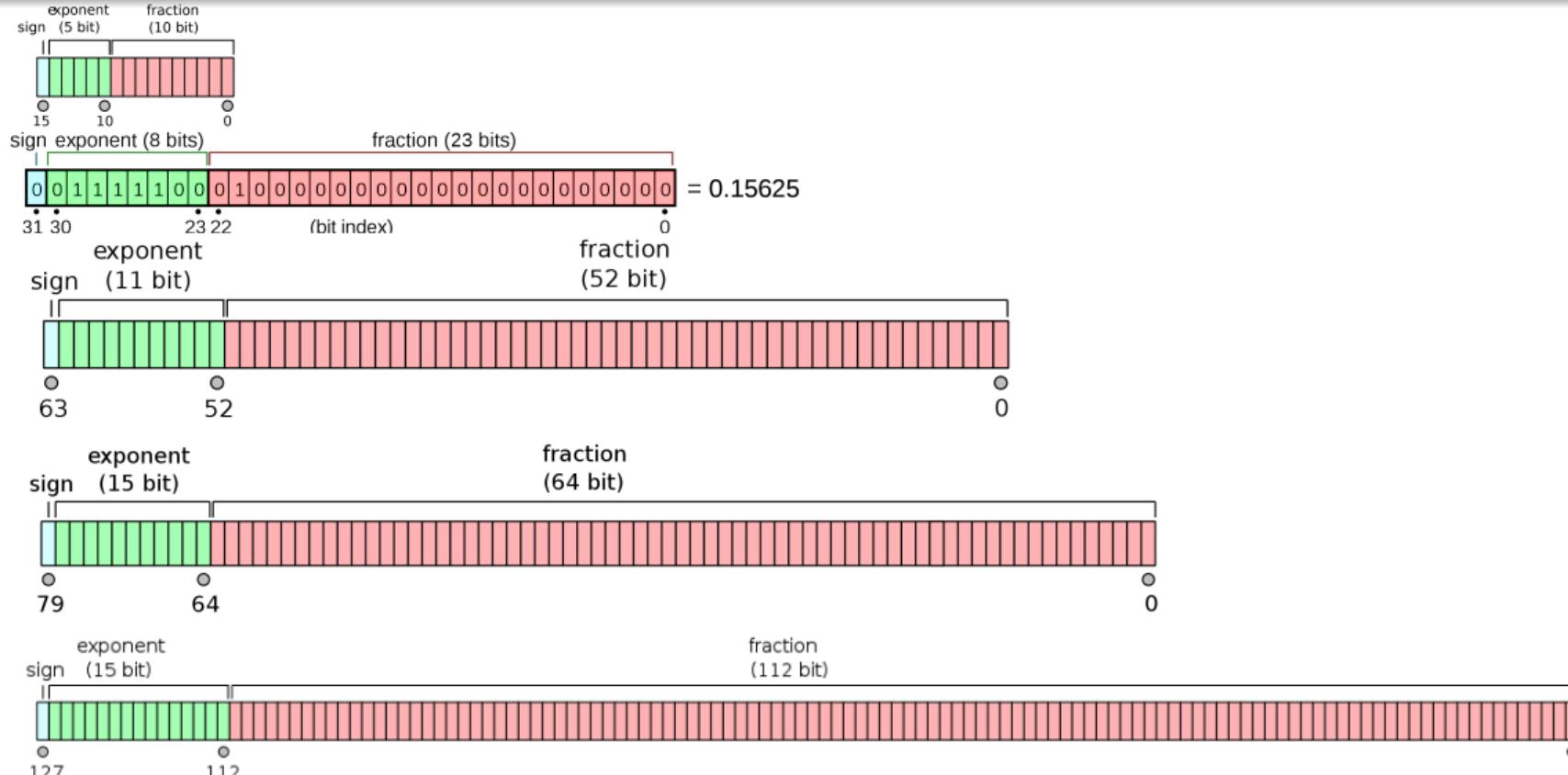
# Representación en Punto Flotante

- Mantisa y exponente pueden representarse:
  - 1 con signo.
  - 2 sin signo.
  - 3 con notación complemento.
  - 4 con notación exceso m.
- Para que las representaciones sean únicas, la mantisa deberá estar normalizada.

## Punto Flotante: Formato IEEE 754

- IEEE (Institute of Electrical and Electronic Engineers).
- El Standard IEEE 754 para punto flotante binario es el mas ampliamente utilizado. En este Standard se especifican los formatos para 32 bits, 64 bits, y 80 bits.
- En 2008 se introdujeron un formato de 16 bits y el de 80 fue reemplazado por uno de 128 bits (IEEE 754-2008).

# Formatos IEEE 754



# IEEE 754: Rangos

Data Type	Length	Precision (Bits)	Approximate Normalized Range	
			Binary	Decimal
Single Precision	32	24	$2^{-126}$ to $2^{127}$	$1.18 \times 10^{-38}$ to $3.40 \times 10^{38}$
Double Precision	64	53	$2^{-1022}$ to $2^{1023}$	$2.23 \times 10^{-308}$ to $1.79 \times 10^{308}$
Double Extended Precision	80	64	$2^{-16382}$ to $2^{16383}$	$3.37 \times 10^{-4932}$ to $1.18 \times 10^{4932}$

1 Fundamentos

2 Procesamiento de Señales digitales

3 Modelo de ejecución SIMD

## 4 Implementaciones SIMD en x86

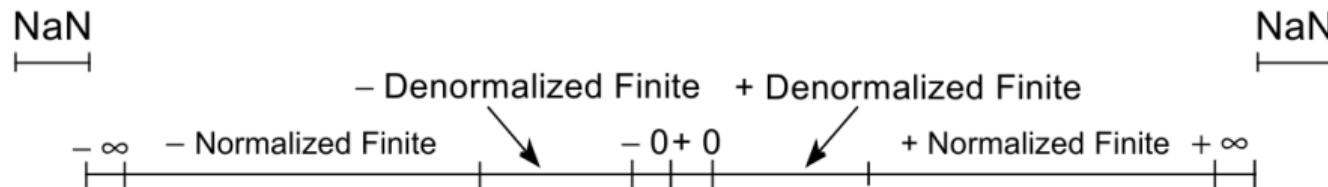
- Arquitectura completa
- Números Reales
- Formatos de Punto Fijo
- Formatos de Punto Flotante
- Codificación de Números Reales

5 Instrucciones

# Codificación de los números normalizados... y los demás

- Ceros signados
- Números finitos de-normalizados
- Números finitos normalizados
- Infinitos signados
- NaNs (Not a Number)
- Números Indefinidos

# Códigos para cada caso



**Real Number and NaN Encodings For 32-Bit Floating-Point Format**

S	E	Sig <sup>1</sup>		S	E	Sig <sup>1</sup>
1	0	0.000...	- 0	+ 0	0	0.000...
1	0	0.XXX... <sup>2</sup>	- Denormalized Finite	+Denormalized Finite	0	0.000...
1	1...254	1.XXX...	- Normalized Finite	+Normalized Finite	0	1...254 1.XXX...
1	255	1.000...	- $\infty$	+ $\infty$	0	255 1.000...
X <sup>3</sup>	255	1.0XX... <sup>2</sup>	SNaN	SNaN	X <sup>3</sup>	255 1.0XX... <sup>2</sup>
X <sup>3</sup>	255	1.1XX...	QNaN	QNaN	X <sup>3</sup>	255 1.1XX...

NOTAS:

# Ceros Signados

# Ceros Signados

- Una operación puede dar +0 o -0 en función del bit de signo.

# Ceros Signados

- Una operación puede dar +0 o -0 en función del bit de signo.
- En ambos casos el valor es el mismo.

# Ceros Signados

- Una operación puede dar +0 o -0 en función del bit de signo.
- En ambos casos el valor es el mismo.
- El signo de un resultado cero depende de la operación en sí y del modo de redondeo utilizado.

# Ceros Signados

- Una operación puede dar +0 o -0 en función del bit de signo.
- En ambos casos el valor es el mismo.
- El signo de un resultado cero depende de la operación en sí y del modo de redondeo utilizado.
- Los ceros signados ayudan a interpretar el intervalo aritmético en el que se ubicaría el resultado si la precisión aritmética fuese mayor.

# Ceros Signados

- Una operación puede dar +0 o -0 en función del bit de signo.
- En ambos casos el valor es el mismo.
- El signo de un resultado cero depende de la operación en sí y del modo de redondeo utilizado.
- Los ceros signados ayudan a interpretar el intervalo aritmético en el que se ubicaría el resultado si la precisión aritmética fuese mayor.
- Indica la dirección desde la cual ocurrió el redondeo a cero, o el signo de un infinito que fue invertido.

# Números Finitos Normalizados

# Números Finitos Normalizados

- El rango de éstos números se compone de todos los valores finitos distintos de cero codificables en formato de números reales entre 0 y  $\pm\infty$ .

# Números Finitos Normalizados

- El rango de éstos números se compone de todos los valores finitos distintos de cero codificables en formato de números reales entre 0 y  $\pm\infty$ .
- En el formato de punto flotante simple precisión estos números se componen de todos aquellos cuyos exponentes desplazados van de 1 a 254, (no desplazados van de -126 a 127).

# Números Finitos Normalizados

- El rango de éstos números se compone de todos los valores finitos distintos de cero codificables en formato de números reales entre 0 y  $\pm\infty$ .
- En el formato de punto flotante simple precisión estos números se componen de todos aquellos cuyos exponentes desplazados van de 1 a 254, (no desplazados van de -126 a 127).
- Cuando se aproximan a cero, estos números no pueden seguir expresándose en este formato, ya que el rango del exponente no puede compensar el desplazamiento a izquierda del punto decimal.

# Números Finitos Normalizados

- El rango de éstos números se compone de todos los valores finitos distintos de cero codificables en formato de números reales entre 0 y  $\pm\infty$ .
- En el formato de punto flotante simple precisión estos números se componen de todos aquellos cuyos exponentes desplazados van de 1 a 254, (no desplazados van de -126 a 127).
- Cuando se aproximan a cero, estos números no pueden seguir expresándose en este formato, ya que el rango del exponente no puede compensar el desplazamiento a izquierda del punto decimal.
- Cuando se llega a un exponente cero en un número normalizado, se pasa al rango de-normalizado.

# Números Finitos Normalizados

# Números Finitos Normalizados

- En general las operaciones entre números normalizados arrojan como resultado otro número normalizado.

# Números Finitos Normalizados

- En general las operaciones entre números normalizados arrojan como resultado otro número normalizado.
- Si hay underflow se pasa a trabajar en formato de-normalizado.

# Números Finitos Normalizados

- En general las operaciones entre números normalizados arrojan como resultado otro número normalizado.
- Si hay underflow se pasa a trabajar en formato de-normalizado.
- Para el resultado de una operación que requiere como exponente -129, tenemos el siguiente proceso de denormalización para representarlo con exponente -126 (extremo inferior del rango).

# Números Finitos Normalizados

- En general las operaciones entre números normalizados arrojan como resultado otro número normalizado.
- Si hay underflow se pasa a trabajar en formato de-normalizado.
- Para el resultado de una operación que requiere como exponente -129, tenemos el siguiente proceso de denormalización para representarlo con exponente -126 (extremo inferior del rango).

<b>Operation</b>	<b>Sign</b>	<b>Exponent*</b>	<b>Significand</b>
True Result	0	-129	1.01011100000...00
Denormalize	0	-128	0.10101110000...00
Denormalize	0	-127	0.01010111000...00
Denormalize	0	-126	0.00101011100...00
Denormal Result	0	-126	0.00101011100...00

# Números Finitos Normalizados

- En general las operaciones entre números normalizados arrojan como resultado otro número normalizado.
- Si hay underflow se pasa a trabajar en formato de-normalizado.
- Para el resultado de una operación que requiere como exponente -129, tenemos el siguiente proceso de denormalización para representarlo con exponente -126 (extremo inferior del rango).

<b>Operation</b>	<b>Sign</b>	<b>Exponent*</b>	<b>Significand</b>
True Result	0	-129	1.01011100000...00
Denormalize	0	-128	0.10101110000...00
Denormalize	0	-127	0.01010111000...00
Denormalize	0	-126	0.00101011100...00
Denormal Result	0	-126	0.00101011100...00

- Observar como se pierden los tres bits menos significativos de la mantisa.

# Sobre infinitos y demás “cosas raras”

# Sobre infinitos y demás “cosas raras”

- Infinitos signados

# Sobre infinitos y demás “cosas raras”

- Infinitos signados

- $-\infty$  y  $+\infty$ , representan los máximos números reales positivo y negativo representables en formato de punto flotante.

# Sobre infinitos y demás “cosas raras”

- Infinitos signados

- $-\infty$  y  $+\infty$ , representan los máximos números reales positivo y negativo representables en formato de punto flotante.
- La mantisa siempre es 1.000....00, y el máximo exponente desplazado representable (p. Ej. 255 para precisión simple).

# Sobre infinitos y demás “cosas raras”

- Infinitos signados

- $-\infty$  y  $+\infty$ , representan los máximos números reales positivo y negativo representables en formato de punto flotante.
- La mantisa siempre es 1.000....00, y el máximo exponente desplazado representable (p. Ej. 255 para precisión simple).

- NaNs

# Sobre infinitos y demás “cosas raras”

- Infinitos signados

- $-\infty$  y  $+\infty$ , representan los máximos números reales positivo y negativo representables en formato de punto flotante.
- La mantisa siempre es 1.000....00, y el máximo exponente desplazado representable (p. Ej. 255 para precisión simple).

- NaNs

- **NaN = Not a Number.**

# Sobre infinitos y demás “cosas raras”

- Infinitos signados
  - $-\infty$  y  $+\infty$ , representan los máximos números reales positivo y negativo representables en formato de punto flotante.
  - La mantisa siempre es 1.000....00, y el máximo exponente desplazado representable (p. Ej. 255 para precisión simple).

- NaNs
  - NaN = **Not a Number**.
  - No son parte del rango de números reales.

# Sobre infinitos y demás “cosas raras”

- Infinitos signados
  - $-\infty$  y  $+\infty$ , representan los máximos números reales positivo y negativo representables en formato de punto flotante.
  - La mantisa siempre es 1.000....00, y el máximo exponente desplazado representable (p. Ej. 255 para precisión simple).

- NaNs
  - **NaN** = **Not a Number**.
  - No son parte del rango de números reales.
  - **QNaN**: Quiet NaN tiene el bit mas significativo fraccional seteado. Pueden propagarse por posteriores operaciones sin generar una excepción.

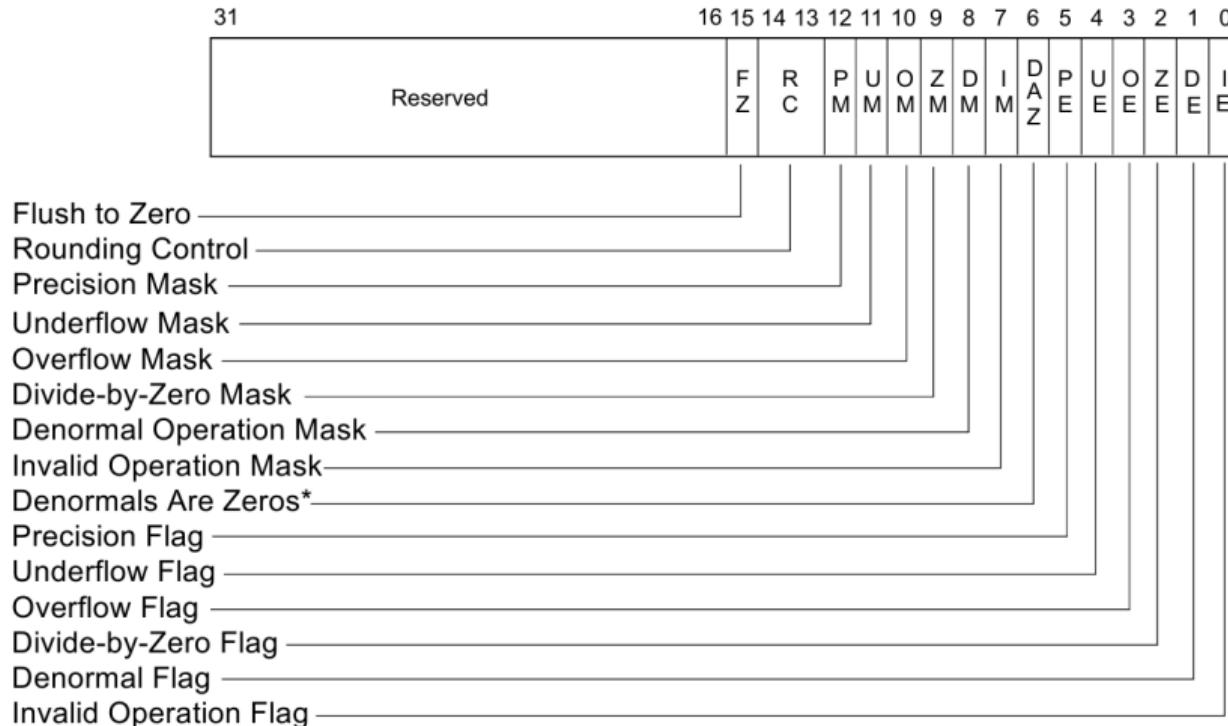
# Sobre infinitos y demás “cosas raras”

- Infinitos signados
  - $-\infty$  y  $+\infty$ , representan los máximos números reales positivo y negativo representables en formato de punto flotante.
  - La mantisa siempre es 1.000....00, y el máximo exponente desplazado representable (p. Ej. 255 para precisión simple).

- NaNs
  - **NaN** = **Not a Number**.
  - No son parte del rango de números reales.
  - **QNaN**: Quiet NaN tiene el bit mas significativo fraccional seteado. Pueden propagarse por posteriores operaciones sin generar una excepción.
  - **SNaN**: Signaled NaN. Tiene en cero el bit fraccional mas significativo. Resulta de una operación inválida de punto flotante.

# Información de control y estado

- Con las extensiones SSE, presentadas en el Pentium III, Intel incluye el registro **MXCSR**.



\* The denormals-are-zeros flag was introduced in the Pentium 4 and Intel Xeon processor.

# Registro MXCSR

# Registro MXCSR

- El bit 15 (FZ) del registro **MXCSR** habilita el modo “flush-to-zero”, que controla la respuesta a la condición de underflow de una instrucción SIMD de punto flotante.

# Registro MXCSR

- El bit 15 (FZ) del registro **MXCSR** habilita el modo “flush-to-zero”, que controla la respuesta a la condición de underflow de una instrucción SIMD de punto flotante.
- Cuando se enmascara la excepción de underflow excepción y se habilita el modo “flush-to-zero”, el procesador realiza las siguientes operaciones cuando detecta una condición de underflow en punto flotante:

# Registro MXCSR

- El bit 15 (FZ) del registro **MXCSR** habilita el modo “flush-to-zero”, que controla la respuesta a la condición de underflow de una instrucción SIMD de punto flotante.
- Cuando se enmascara la excepción de underflow excepción y se habilita el modo “flush-to-zero”, el procesador realiza las siguientes operaciones cuando detecta una condición de underflow en punto flotante:
  - Retorna cero con el signo del resultado correcto.

# Registro MXCSR

- El bit 15 (FZ) del registro **MXCSR** habilita el modo “flush-to-zero”, que controla la respuesta a la condición de underflow de una instrucción SIMD de punto flotante.
- Cuando se enmascara la excepción de underflow excepción y se habilita el modo “flush-to-zero”, el procesador realiza las siguientes operaciones cuando detecta una condición de underflow en punto flotante:
  - Retorna cero con el signo del resultado correcto.
  - Pone en '1' los flags de las excepciones de precisión y de underflow.

# Registro MXCSR

# Registro MXCSR

- Si no se enmascara la excepción de underflow, se ignora el bit “flush-to-zero”. El modo “flush-to-zero” no es compatible con el Standard IEEE 754.

# Registro MXCSR

- Si no se enmascara la excepción de underflow, se ignora el bit “flush-to-zero”. El modo “flush-to-zero” no es compatible con el Standard IEEE 754.
- La respuesta indicada por el IEEE a un underflow es entregar el resultado denormalizado.

# Registro MXCSR

- Si no se enmascara la excepción de underflow, se ignora el bit “flush-to-zero”. El modo “flush-to-zero” no es compatible con el Standard IEEE 754.
- La respuesta indicada por el IEEE a un underflow es entregar el resultado denormalizado.
- El modo “flush-to-zero” se provee en principio por razones de performance.

# Registro MXCSR

- Si no se enmascara la excepción de underflow, se ignora el bit “flush-to-zero”. El modo “flush-to-zero” no es compatible con el Standard IEEE 754.
- La respuesta indicada por el IEEE a un underflow es entregar el resultado denormalizado.
- El modo “flush-to-zero” se provee en principio por razones de performance.
- Al costo de alguna pérdida de precisión, puede obtenerse una ejecución mas rápida para aplicaciones en donde la condición de underflow es común y es tolerable el redondeo a cero cuando se produce el underflow.

# Registro MXCSR

- Si no se enmascara la excepción de underflow, se ignora el bit “flush-to-zero”. El modo “flush-to-zero” no es compatible con el Standard IEEE 754.
- La respuesta indicada por el IEEE a un underflow es entregar el resultado denormalizado.
- El modo “flush-to-zero” se provee en principio por razones de performance.
- Al costo de alguna pérdida de precisión, puede obtenerse una ejecución mas rápida para aplicaciones en donde la condición de underflow es común y es tolerable el redondeo a cero cuando se produce el underflow.
- El bit “flush-to-zero” se limpia durante el arranque o en el reset del procesador, y se deshabilita el modo “flush-to-zero”.

1 Fundamentos

2 Procesamiento de Señales digitales

3 Modelo de ejecución SIMD

4 Implementaciones SIMD en x86

## 5 Instrucciones

- Transferencias (las mas comunes)
- Aritmética en algoritmos DSP
- Instrucciones de punto flotante
- Instrucciones para manejo de enteros para SSEn
- Instrucciones para manejo de cacheabilidad

# Instrucciones de movimiento

MOV <b>D</b>	MOV <b>Q</b>	Move Doubleword/Quadword
MOV <b>SS</b>	MOV <b>SD</b>	Moves a 32bits Single FP/64bits Double FP
MOV <b>DQA</b>	MOV <b>DQU</b>	Moves aligned/unaligned double quadword
MOV <b>APS</b>	MOV <b>UPS</b>	Moves 4 aligned/unaligned 32bit singles
MOV <b>APD</b>	MOV <b>UPD</b>	Moves 2 aligned/unaligned 64bit doubles

# Instrucciones de movimiento

MOV <b>D</b>	MOV <b>Q</b>	Move Doubleword/Quadword
MOV <b>SS</b>	MOV <b>SD</b>	Moves a 32bits Single FP/64bits Double FP
MOV <b>DQA</b>	MOV <b>DQU</b>	Moves aligned/unaligned double quadword
MOV <b>APS</b>	MOV <b>UPS</b>	Moves 4 aligned/unaligned 32bit singles
MOV <b>APD</b>	MOV <b>UPD</b>	Moves 2 aligned/unaligned 64bit doubles

Ejemplo:



# Instrucciones de movimiento

MOV <b>D</b>	MOV <b>Q</b>	Move Doubleword/Quadword
MOV <b>SS</b>	MOV <b>SD</b>	Moves a 32bits Single FP/64bits Double FP
MOV <b>DQA</b>	MOV <b>DQU</b>	Moves aligned/unaligned double quadword
MOV <b>APS</b>	MOV <b>UPS</b>	Moves 4 aligned/unaligned 32bit singles
MOV <b>APD</b>	MOV <b>UPD</b>	Moves 2 aligned/unaligned 64bit doubles

Ejemplo:

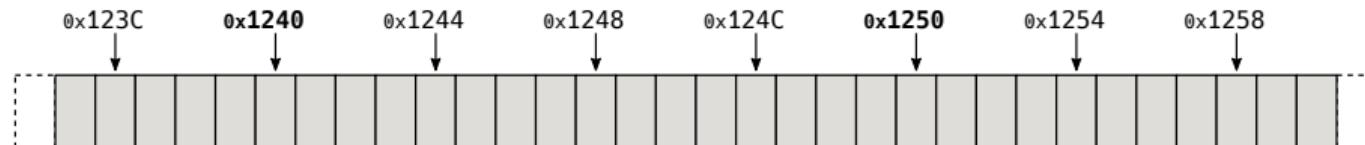


MOV**D** [0x123C], xmm0      ✓      [0x123C] ← xmm0(32:0)

# Instrucciones de movimiento

MOV <b>D</b>	MOV <b>Q</b>	Move Doubleword/Quadword
MOV <b>SS</b>	MOV <b>SD</b>	Moves a 32bits Single FP/64bits Double FP
MOV <b>DQA</b>	MOV <b>DQU</b>	Moves aligned/unaligned double quadword
MOV <b>APS</b>	MOV <b>UPS</b>	Moves 4 aligned/unaligned 32bit singles
MOV <b>APD</b>	MOV <b>UPD</b>	Moves 2 aligned/unaligned 64bit doubles

Ejemplo:

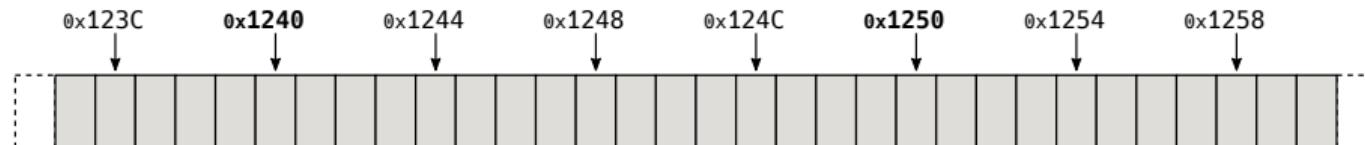


MOV**D** [0x123C], xmm0      ✓      [0x123C] ← xmm0(32:0)  
MOV**Q** xmm0, [0x1245]      ✓      xmm0(64:0) ← [0x1245]

# Instrucciones de movimiento

MOV <b>D</b>	MOV <b>Q</b>	Move Doubleword/Quadword
MOV <b>SS</b>	MOV <b>SD</b>	Moves a 32bits Single FP/64bits Double FP
MOV <b>DQA</b>	MOV <b>DQU</b>	Moves aligned/unaligned double quadword
MOV <b>APS</b>	MOV <b>UPS</b>	Moves 4 aligned/unaligned 32bit singles
MOV <b>APD</b>	MOV <b>UPD</b>	Moves 2 aligned/unaligned 64bit doubles

Ejemplo:

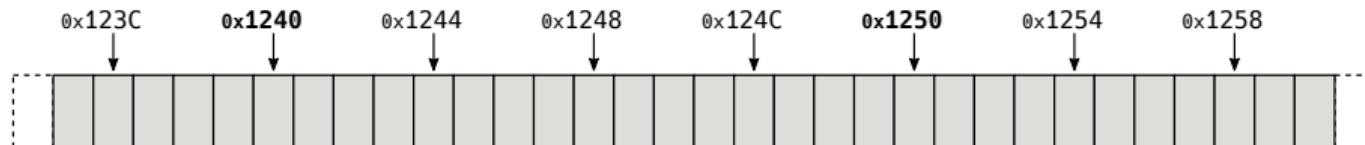


- |                       |   |                              |
|-----------------------|---|------------------------------|
| MOVD [0x123C], xmm0   | ✓ | [0x123C] ← xmm0(32:0)        |
| MOVQ xmm0, [0x1245]   | ✓ | xmm0(64:0) ← [0x1245]        |
| MOVDQA xmm0, [0x1245] | ✗ | Error dirección no alineada.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

# Instrucciones de movimiento

<b>MOV</b> D	<b>MOV</b> Q	Move Doubleword/Quadword
<b>MOV</b> SS	<b>MOV</b> SD	Moves a 32bits Single FP/64bits Double FP
<b>MOV</b> DQA	<b>MOV</b> DQU	Moves aligned/unaligned double quadword
<b>MOV</b> APS	<b>MOV</b> UPS	Moves 4 aligned/unaligned 32bit singles
<b>MOV</b> APD	<b>MOV</b> UPD	Moves 2 aligned/unaligned 64bit doubles

Ejemplo:

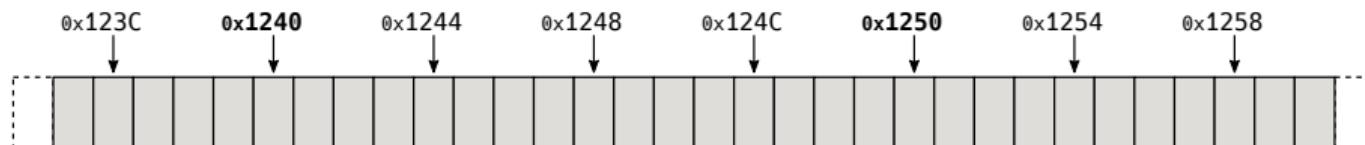


- |                       |   |                              |
|-----------------------|---|------------------------------|
| MOVD [0x123C], xmm0   | ✓ | [0x123C] ← xmm0(32:0)        |
| MOVQ xmm0, [0x1245]   | ✓ | xmm0(64:0) ← [0x1245]        |
| MOVDQA xmm0, [0x1245] | ✗ | Error dirección no alineada.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

# Instrucciones de movimiento

MOV <b>D</b>	MOV <b>Q</b>	Move Doubleword/Quadword
MOV <b>SS</b>	MOV <b>SD</b>	Moves a 32bits Single FP/64bits Double FP
MOV <b>DQA</b>	MOV <b>DQU</b>	Moves aligned/unaligned double quadword
MOV <b>APS</b>	MOV <b>UPS</b>	Moves 4 aligned/unaligned 32bit singles
MOV <b>APD</b>	MOV <b>UPD</b>	Moves 2 aligned/unaligned 64bit doubles

Ejemplo:

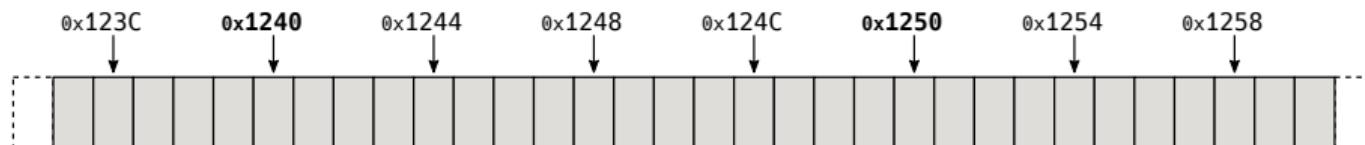


MOVD [0x123C], xmm0	✓	[0x123C] ← xmm0(32:0)
MOVQ xmm0, [0x1245]	✓	xmm0(64:0) ← [0x1245]
MOVDQA xmm0, [0x1245]	✗	Error dirección no alineada.
MOVDQA [0x1250], xmm0	✓	[0x1250] ← xmm0(128:0)
MOVSS xmm0, [0x1248]	✓	xmm0(32:0) ← [0x1248]; sobre punto flotante

# Instrucciones de movimiento

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

Ejemplo:



MOVD [0x123C], xmm0	✓	[0x123C] ← xmm0(32:0)
MOVQ xmm0, [0x1245]	✓	xmm0(64:0) ← [0x1245]
MOV DQA xmm0, [0x1245]	✗	Error dirección no alineada.
MOV DQA [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

PMOV <b>SXBW</b>	PMOV <b>ZXBW</b>	packed sign/zero extension byte to word
PMOV <b>SXBD</b>	PMOV <b>ZXBD</b>	packed sign/zero extension byte to dword
PMOV <b>SXBQ</b>	PMOV <b>ZXBQ</b>	packed sign/zero extension byte to qword
PMOV <b>SXWD</b>	PMOV <b>ZXWD</b>	packed sign/zero extension word to dword
PMOV <b>SXWQ</b>	PMOV <b>ZXWQ</b>	packed sign/zero extension word to qword
PMOV <b>SXDQ</b>	PMOV <b>ZXDQ</b>	packed sign/zero extension word to dqword

# Operaciones Load/Store

PMOV <b>SXBW</b>	PMOV <b>ZXBW</b>	packed sign/zero extension byte to word
PMOV <b>SXBD</b>	PMOV <b>ZXBD</b>	packed sign/zero extension byte to dword
PMOV <b>SXBQ</b>	PMOV <b>ZXBQ</b>	packed sign/zero extension byte to qword
PMOV <b>SXWD</b>	PMOV <b>ZXWD</b>	packed sign/zero extension word to dword
PMOV <b>SXWQ</b>	PMOV <b>ZXWQ</b>	packed sign/zero extension word to qword
PMOV <b>SXDQ</b>	PMOV <b>ZXDQ</b>	packed sign/zero extension word to dqword

# Operaciones Load/Store

PMOV <b>SXBW</b>	PMOV <b>ZXBW</b>	packed sign/zero extension byte to word
PMOV <b>SXBD</b>	PMOV <b>ZXBD</b>	packed sign/zero extension byte to dword
PMOV <b>SXBQ</b>	PMOV <b>ZXBQ</b>	packed sign/zero extension byte to qword
PMOV <b>SXWD</b>	PMOV <b>ZXWD</b>	packed sign/zero extension word to dword
PMOV <b>SXWQ</b>	PMOV <b>ZXWQ</b>	packed sign/zero extension word to qword
PMOV <b>SXDQ</b>	PMOV <b>ZXDQ</b>	packed sign/zero extension word to dqword

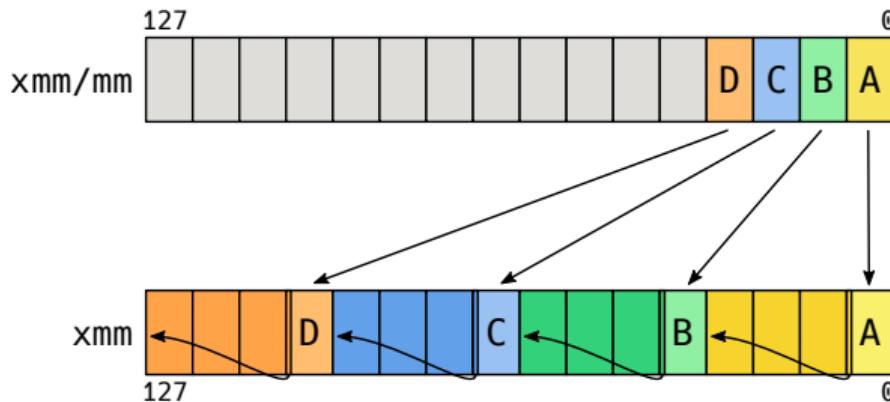
Ejemplos:

# Operaciones Load/Store

PMOV <b>SX</b> BW	PMOV <b>ZX</b> BW	packed sign/zero extension byte to word
PMOV <b>SX</b> BD	PMOV <b>ZX</b> BD	packed sign/zero extension byte to dword
PMOV <b>SX</b> BQ	PMOV <b>ZX</b> BQ	packed sign/zero extension byte to qword
PMOV <b>SX</b> WD	PMOV <b>ZX</b> WD	packed sign/zero extension word to dword
PMOV <b>SX</b> WQ	PMOV <b>ZX</b> WQ	packed sign/zero extension word to qword
PMOV <b>SX</b> DQ	PMOV <b>ZX</b> DQ	packed sign/zero extension word to dqword

Ejemplos:

PMOV**SX**BD xmm0, xmm0



# Operaciones Load/Store

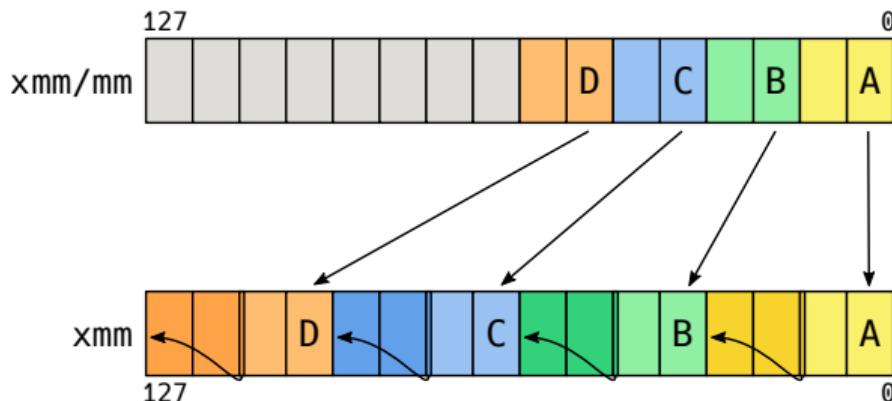
PMOV <b>SX</b> BW	PMOV <b>ZX</b> BW	packed sign/zero extension byte to word
PMOV <b>SX</b> BD	PMOV <b>ZX</b> BD	packed sign/zero extension byte to dword
PMOV <b>SX</b> BQ	PMOV <b>ZX</b> BQ	packed sign/zero extension byte to qword
PMOV <b>SX</b> WD	PMOV <b>ZX</b> WD	packed sign/zero extension word to dword
PMOV <b>SX</b> WQ	PMOV <b>ZX</b> WQ	packed sign/zero extension word to qword
PMOV <b>SX</b> DQ	PMOV <b>ZX</b> DQ	packed sign/zero extension word to dqword

Ejemplos:

PMOV**SX**BD xmm0, xmm0



PMOV**ZX**WD xmm0, [data]

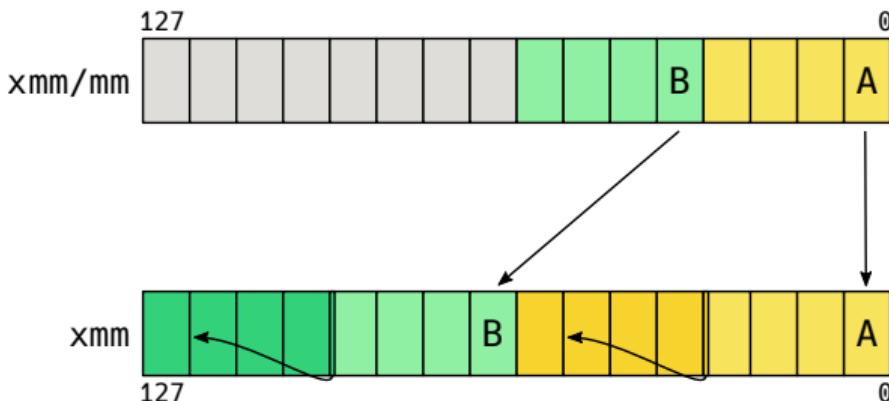


# Operaciones Load/Store

PMOV <b>SX</b> BW	PMOV <b>ZX</b> BW	packed sign/zero extension byte to word
PMOV <b>SX</b> BD	PMOV <b>ZX</b> BD	packed sign/zero extension byte to dword
PMOV <b>SX</b> BQ	PMOV <b>ZX</b> BQ	packed sign/zero extension byte to qword
PMOV <b>SX</b> WD	PMOV <b>ZX</b> WD	packed sign/zero extension word to dword
PMOV <b>SX</b> WQ	PMOV <b>ZX</b> WQ	packed sign/zero extension word to qword
PMOV <b>SX</b> DQ	PMOV <b>ZX</b> DQ	packed sign/zero extension word to dqword

Ejemplos:

- |                                |   |
|--------------------------------|---|
| PMOV <b>SX</b> BD xmm0, xmm0   | ✓ |
| PMOV <b>ZX</b> WD xmm0, [data] | ✓ |
| PMOV <b>ZX</b> DQ xmm0, xmm1   | ✓ |



# Operaciones Load/Store

PMOV <b>SX</b> BW	PMOV <b>ZX</b> BW	packed sign/zero extension byte to word
PMOV <b>SX</b> BD	PMOV <b>ZX</b> BD	packed sign/zero extension byte to dword
PMOV <b>SX</b> BQ	PMOV <b>ZX</b> BQ	packed sign/zero extension byte to qword
PMOV <b>SX</b> WD	PMOV <b>ZX</b> WD	packed sign/zero extension word to dword
PMOV <b>SX</b> WQ	PMOV <b>ZX</b> WQ	packed sign/zero extension word to qword
PMOV <b>SX</b> DQ	PMOV <b>ZX</b> DQ	packed sign/zero extension word to dqword

Ejemplos:

PMOV**SX**BD xmm0, xmm0 ✓

PMOV**ZX**WD xmm0, [data] ✓

PMOV**ZX**DQ xmm0, xmm1 ✓

PMOV**ZX**QD xmm0, xmm0 ✗ Instrucción invalida.

# Operaciones Load/Store

PMOV <b>SXBW</b>	PMOV <b>ZXBW</b>	packed sign/zero extension byte to word
PMOV <b>SXBD</b>	PMOV <b>ZXBD</b>	packed sign/zero extension byte to dword
PMOV <b>SXBQ</b>	PMOV <b>ZXBQ</b>	packed sign/zero extension byte to qword
PMOV <b>SXWD</b>	PMOV <b>ZXWD</b>	packed sign/zero extension word to dword
PMOV <b>SXWQ</b>	PMOV <b>ZXWQ</b>	packed sign/zero extension word to qword
PMOV <b>SXDQ</b>	PMOV <b>ZXDQ</b>	packed sign/zero extension word to dqword

Ejemplos:

PMOV <b>SXBD</b> xmm0, xmm0	✓
PMOV <b>ZXWD</b> xmm0, [data]	✓
PMOV <b>ZXDQ</b> xmm0, xmm1	✓
PMOV <b>ZXQD</b> xmm0, xmm0	✗ Instrucción invalida.
PMOV <b>SXBD</b> [data], xmm0	✗ Modo de direccionamiento invalido.

1 Fundamentos

2 Procesamiento de Señales digitales

3 Modelo de ejecución SIMD

4 Implementaciones SIMD en x86

## 5 Instrucciones

- Transferencias (las mas comunes)
- **Aritmética en algoritmos DSP**
- Instrucciones de punto flotante
- Instrucciones para manejo de enteros para SSEn
- Instrucciones para manejo de cacheabilidad

# Acumulación de resultados sobre un registro

# Acumulación de resultados sobre un registro

- A medida que acumulamos resultados en un registro, llegamos inexorablemente al punto en el que el rango del resultado excede la capacidad de bits del registro de acumulación.

# Acumulación de resultados sobre un registro

- A medida que acumulamos resultados en un registro, llegamos inexorablemente al punto en el que el rango del resultado excede la capacidad de bits del registro de acumulación.
- En este punto los procesadores convencionales indican la situación mediante un flag de overflow, y en el operando destino se almacena el valor de **desborde**.

# Acumulación de resultados sobre un registro

- A medida que acumulamos resultados en un registro, llegamos inexorablemente al punto en el que el rango del resultado excede la capacidad de bits del registro de acumulación.
- En este punto los procesadores convencionales indican la situación mediante un flag de overflow, y en el operando destino se almacena el valor de **desborde**.
- La aplicación chequea dentro del lazo de cálculo este flag y de acuerdo a su estado decide si sigue la acumulación.

# Acumulación de resultados sobre un registro

- A medida que acumulamos resultados en un registro, llegamos inexorablemente al punto en el que el rango del resultado excede la capacidad de bits del registro de acumulación.
- En este punto los procesadores convencionales indican la situación mediante un flag de overflow, y en el operando destino se almacena el valor de **desborde**.
- La aplicación chequea dentro del lazo de cálculo este flag y de acuerdo a su estado decide si sigue la acumulación.
- El costo de esta práctica de programación es tiempo de ejecución para analizar como tratar cada resultado parcial.

# Manejo de los desbordes

# Manejo de los desbordes

**Aritmética de Desborde** Del comportamiento planteado en el slide anterior, proviene el nombre que se le ha asignado a la forma de aritmética empleada por los procesadores convencionales: Al llegar al extremo superior del rango realizan una operación de que se denomina wraparound y consiste en resetear el contador al valor inicial y setear el flag de overflow). Si por el contrario se sustrae un registro al llegar a cero, si se pide una nueva sustracción se pasa al valor máximo y se sigue desde allí. El costo es comprobar en cada ciclo de un lazo el valor del flag para decidir si se continúa en el lazo o no.

# Manejo de los desbordes

**Aritmética de Desborde** Del comportamiento planteado en el slide anterior, proviene el nombre que se le ha asignado a la forma de aritmética empleada por los procesadores convencionales: Al llegar al extremo superior del rango realizan una operación de que se denomina wraparound y consiste en resetear el contador al valor inicial y setear el flag de overflow). Si por el contrario se sustrae un registro al llegar a cero, si se pide una nueva sustracción se pasa al valor máximo y se sigue desde allí. El costo es comprobar en cada ciclo de un lazo el valor del flag para decidir si se continúa en el lazo o no.

**Aritmética Saturada** Al producirse una condición fuera de rango el operando destino mantiene el máximo o mínimo valor del rango (dependiendo si la condición se ha producido por exceso o por defecto).

# Aritmética Saturada

Tipo de Dato	Límite Inferior		Límite Superior	
	Hexadecimal	Decimal	Hexadecimal	Decimal
byte signado	0x80	-128	0x7F	127
word signada	0x8000	-32768	0x7FFF	32767

Cuadro: Aritmética Saturada Signada

Tipo de Dato	Límite Inferior		Límite Superior	
	Hexadecimal	Decimal	Hexadecimal	Decimal
byte no signado	0x00	0	0xFF	255
word no signada	0x0000	0	0xFFFF	65535

Cuadro: Aritmética Saturada No Signada

# Aritmética Saturada vs. Aritmética de Desborde

- Consideremos la siguiente suma empaquetada de 8 bytes.
- En el byte 6 puede observarse la diferencia entre ambas operaciones

Byte	7	6	5	4	3	2	1	0
oper1	0x4D	0x23	0x9F	0xC0	0x11	0x4A	0x29	0x0B
oper 2	0x32	0xFF	0x1A	0x0D	0x3F	0xAF	0xB0	0x36
desb.	0x7F	0x22	0xB9	0xCD	0x50	0xF9	0xD9	0x41
sat.	0x7F	0xFF	0xB9	0xCD	0x50	0xF9	0xD9	0x41

Cuadro: Suma Empaquetada saturada vs. suma empaquetada de desborde

# Cuando usar aritmética saturada o de desborde

# Cuando usar aritmética saturada o de desborde

- En algunas aplicaciones la aritmética saturada provee soluciones más eficaces que la aritmética de desborde.

## Cuando usar aritmética saturada o de desborde

- En algunas aplicaciones la aritmética saturada provee soluciones más eficaces que la aritmética de desborde.
- Al procesar video, cuando un nivel de negro satura, no tiene sentido seguir procesando la variable ya que no produce mas efecto sobre la salida.

## Cuando usar aritmética saturada o de desborde

- En algunas aplicaciones la aritmética saturada provee soluciones más eficaces que la aritmética de desborde.
- Al procesar video, cuando un nivel de negro satura, no tiene sentido seguir procesando la variable ya que no produce mas efecto sobre la salida.
- Si utilizamos la habitual lógica de desborde, el valor remanente en el registro de resultado al saturar el negro nos lleva de vuelta al blanco, afectando el flag de overflow para prevenirnos de la situación, pero el número almacenado en el operando del resultado es mucho menor al máximo del rango.

# Operaciones Aritméticas

PADD <b>B</b>	PADD <b>W</b>	PADD <b>D</b>	PADD <b>Q</b>	Add Integer
PSUB <b>B</b>	PSUB <b>W</b>	PSUB <b>D</b>	PSUB <b>Q</b>	Sub Integer
PMUL <b>HW</b>	PMUL <b>LW</b>			Mul Integer Word
PMUL <b>HD</b>	PMUL <b>LD</b>			Mul Integer Dword
PMIN <b>SB</b>	PMAX <b>SB</b>	PMIN <b>UB</b>	PMAX <b>UB</b>	Max and Min Integer
PMIN <b>SW</b>	PMAX <b>SW</b>	PMIN <b>UW</b>	PMAX <b>UW</b>	Max and Min Integer
PMIN <b>SD</b>	PMAX <b>SD</b>	PMIN <b>UD</b>	PMAX <b>UD</b>	Max and Min Integer

# Operaciones Aritméticas

PADD <b>B</b>	PADD <b>W</b>	PADD <b>D</b>	PADD <b>Q</b>	Add Integer
PSUB <b>B</b>	PSUB <b>W</b>	PSUB <b>D</b>	PSUB <b>Q</b>	Sub Integer
PMUL <b>HW</b>	PMUL <b>LW</b>			Mul Integer Word
PMUL <b>HD</b>	PMUL <b>LD</b>			Mul Integer Dword
PMIN <b>SB</b>	PMAX <b>SB</b>	PMIN <b>UB</b>	PMAX <b>UB</b>	Max and Min Integer
PMIN <b>SW</b>	PMAX <b>SW</b>	PMIN <b>UW</b>	PMAX <b>UW</b>	Max and Min Integer
PMIN <b>SD</b>	PMAX <b>SD</b>	PMIN <b>UD</b>	PMAX <b>UD</b>	Max and Min Integer

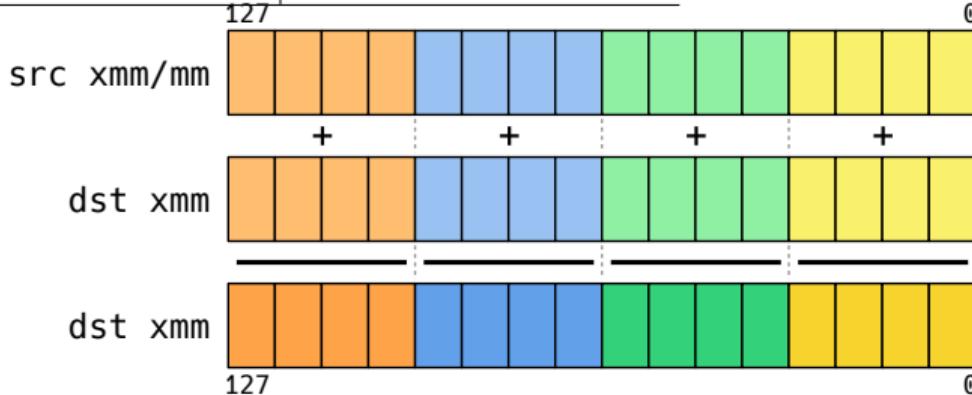
Ejemplos:

# Operaciones Aritméticas

PADD <b>B</b>	PADD <b>W</b>	PADD <b>D</b>	PADD <b>Q</b>	Add Integer
PSUB <b>B</b>	PSUB <b>W</b>	PSUB <b>D</b>	PSUB <b>Q</b>	Sub Integer
PMUL <b>HW</b>	PMUL <b>LW</b>			Mul Integer Word
PMUL <b>HD</b>	PMUL <b>LD</b>			Mul Integer Dword
PMIN <b>SB</b>	PMAX <b>SB</b>	PMIN <b>UB</b>	PMAX <b>UB</b>	Max and Min Integer
PMIN <b>SW</b>	PMAX <b>SW</b>	PMIN <b>UW</b>	PMAX <b>UW</b>	Max and Min Integer
PMIN <b>SD</b>	PMAX <b>SD</b>	PMIN <b>UD</b>	PMAX <b>UD</b>	Max and Min Integer

Ejemplos:

PADDD xmm0, xmm1

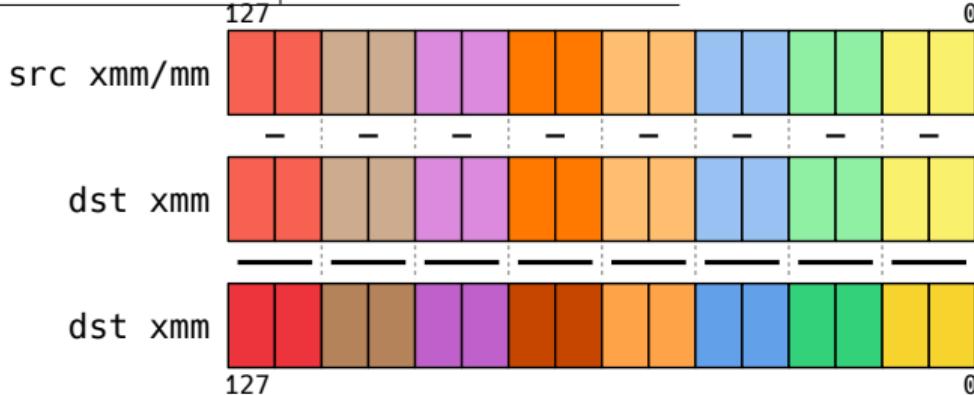


# Operaciones Aritméticas

PADD <b>B</b>	PADD <b>W</b>	PADD <b>D</b>	PADD <b>Q</b>	Add Integer
PSUB <b>B</b>	PSUB <b>W</b>	PSUB <b>D</b>	PSUB <b>Q</b>	Sub Integer
PMUL <b>HW</b>	PMUL <b>LW</b>			Mul Integer Word
PMUL <b>HD</b>	PMUL <b>LD</b>			Mul Integer Dword
PMIN <b>SB</b>	PMAX <b>SB</b>	PMIN <b>UB</b>	PMAX <b>UB</b>	Max and Min Integer
PMIN <b>SW</b>	PMAX <b>SW</b>	PMIN <b>UW</b>	PMAX <b>UW</b>	Max and Min Integer
PMIN <b>SD</b>	PMAX <b>SD</b>	PMIN <b>UD</b>	PMAX <b>UD</b>	Max and Min Integer

Ejemplos:

- PADD **D** xmm0, xmm1 ✓
- PSUB **W** xmm0, [data] ✓

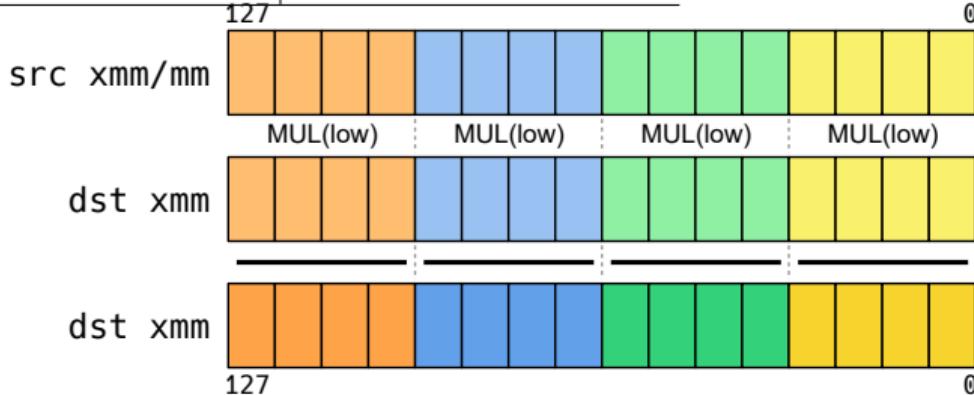


# Operaciones Aritméticas

PADD <b>B</b>	PADD <b>W</b>	PADD <b>D</b>	PADD <b>Q</b>	Add Integer
PSUB <b>B</b>	PSUB <b>W</b>	PSUB <b>D</b>	PSUB <b>Q</b>	Sub Integer
PMUL <b>HW</b>	PMUL <b>LW</b>			Mul Integer Word
PMUL <b>HD</b>	PMUL <b>LD</b>			Mul Integer Dword
PMIN <b>SB</b>	PMAX <b>SB</b>	PMIN <b>UB</b>	PMAX <b>UB</b>	Max and Min Integer
PMIN <b>SW</b>	PMAX <b>SW</b>	PMIN <b>UW</b>	PMAX <b>UW</b>	Max and Min Integer
PMIN <b>SD</b>	PMAX <b>SD</b>	PMIN <b>UD</b>	PMAX <b>UD</b>	Max and Min Integer

Ejemplos:

- PADD D xmm0, xmm1      ✓
- PSUB W xmm0, [data]      ✓
- PMUL LD xmm0, xmm1      ✓

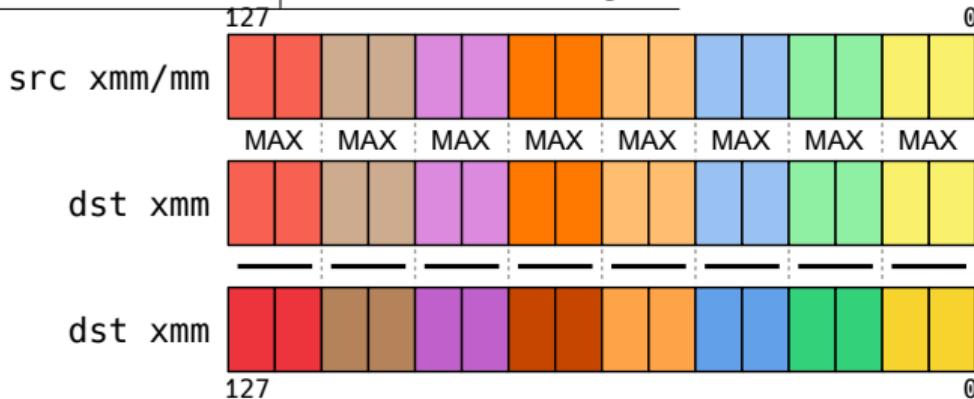


# Operaciones Aritméticas

PADD <b>B</b>	PADD <b>W</b>	PADD <b>D</b>	PADD <b>Q</b>	Add Integer
PSUB <b>B</b>	PSUB <b>W</b>	PSUB <b>D</b>	PSUB <b>Q</b>	Sub Integer
PMUL <b>HW</b>	PMUL <b>LW</b>			Mul Integer Word
PMUL <b>HD</b>	PMUL <b>LD</b>			Mul Integer Dword
PMIN <b>SB</b>	PMAX <b>SB</b>	PMIN <b>UB</b>	PMAX <b>UB</b>	Max and Min Integer
PMIN <b>SW</b>	PMAX <b>SW</b>	PMIN <b>UW</b>	PMAX <b>UW</b>	Max and Min Integer
PMIN <b>SD</b>	PMAX <b>SD</b>	PMIN <b>UD</b>	PMAX <b>UD</b>	Max and Min Integer

Ejemplos:

- |                             |   |
|-----------------------------|---|
| PADD <b>D</b> xmm0, xmm1    | ✓ |
| PSUB <b>W</b> xmm0, [data]  | ✓ |
| PMUL <b>LD</b> xmm0, xmm1   | ✓ |
| PMAX <b>SW</b> xmm0, [data] | ✓ |



# Operaciones Aritméticas

PADD <b>B</b>	PADD <b>W</b>	PADD <b>D</b>	PADD <b>Q</b>	Add Integer
PSUB <b>B</b>	PSUB <b>W</b>	PSUB <b>D</b>	PSUB <b>Q</b>	Sub Integer
PMUL <b>HW</b>	PMUL <b>LW</b>			Mul Integer Word
PMUL <b>HD</b>	PMUL <b>LD</b>			Mul Integer Dword
PMIN <b>SB</b>	PMAX <b>SB</b>	PMIN <b>UB</b>	PMAX <b>UB</b>	Max and Min Integer
PMIN <b>SW</b>	PMAX <b>SW</b>	PMIN <b>UW</b>	PMAX <b>UW</b>	Max and Min Integer
PMIN <b>SD</b>	PMAX <b>SD</b>	PMIN <b>UD</b>	PMAX <b>UD</b>	Max and Min Integer

Ejemplos:

- |                     |   |
|---------------------|---|
| PADD D 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

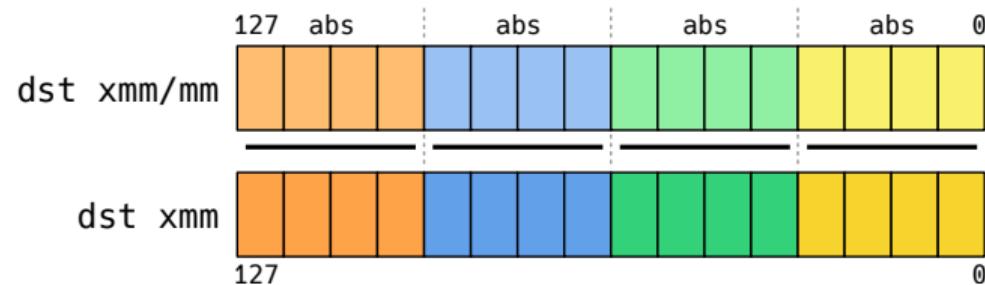
Ejemplos:

# Operaciones Aritméticas

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

Ejemplos:

PABSD xmm0, xmm0

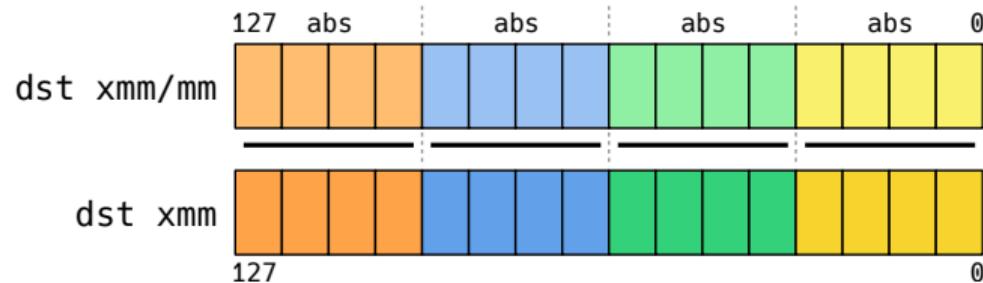


# Operaciones Aritméticas

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]

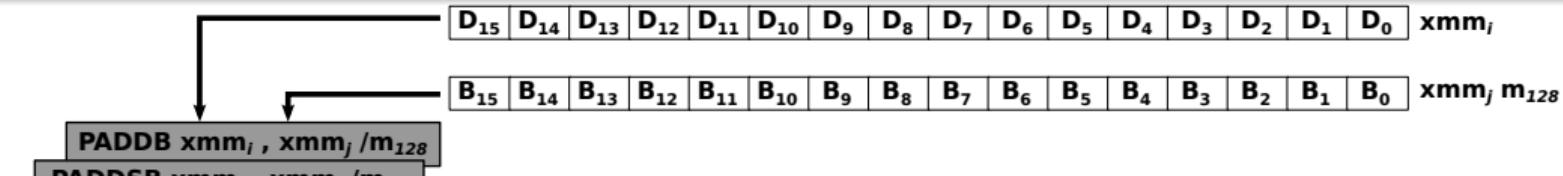
✓

PABSD [data], xmm0

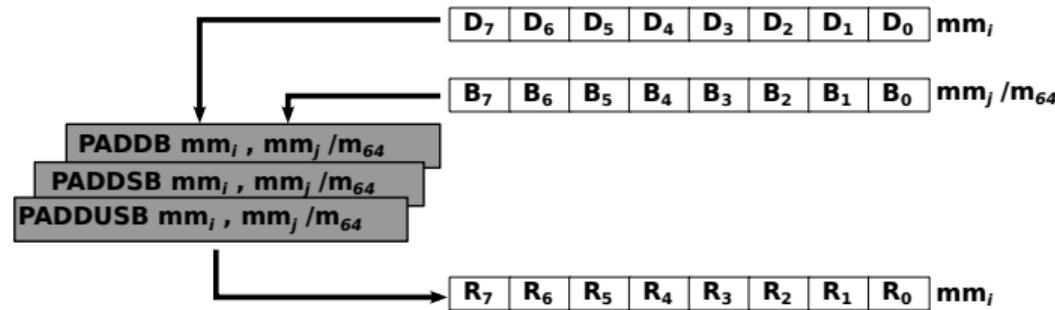
✗

Modo de direccionamiento invalido.

# Suma empaquetada de bytes

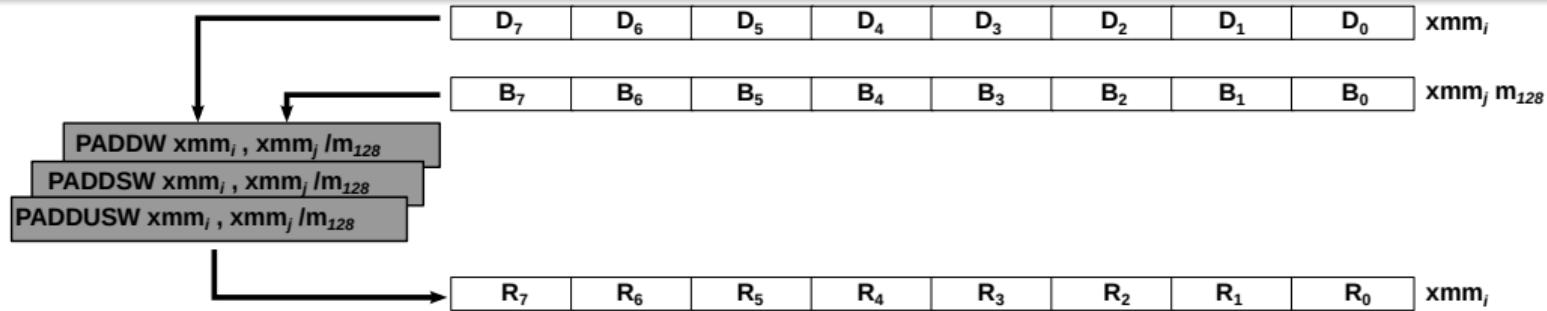


$$R_k = D_k + B_k$$

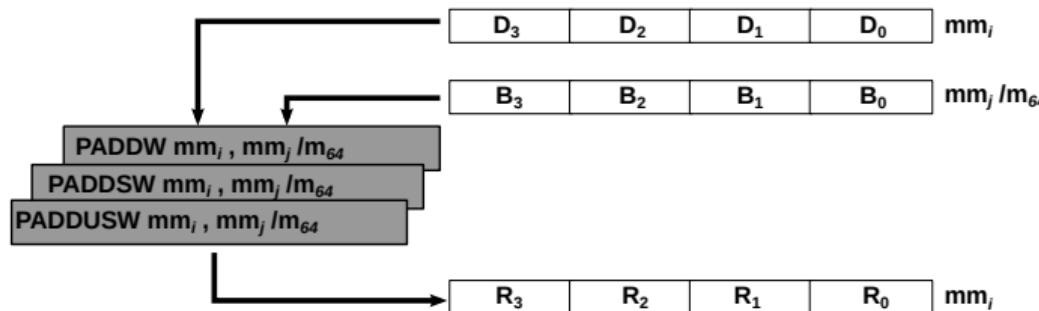


$$R_k = D_k + B_k$$

# Suma empaquetada de words

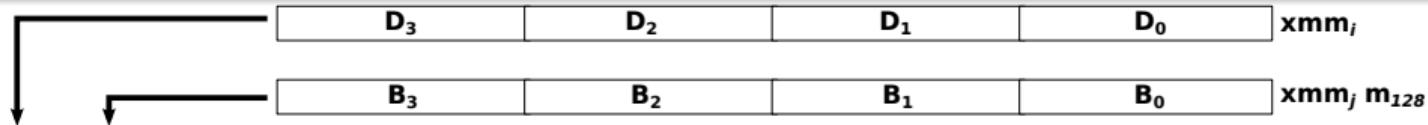


$$R_k = D_k + B_k$$

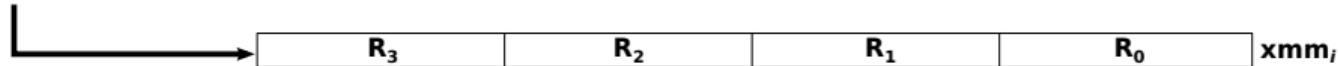


$$R_k = D_k + B_k$$

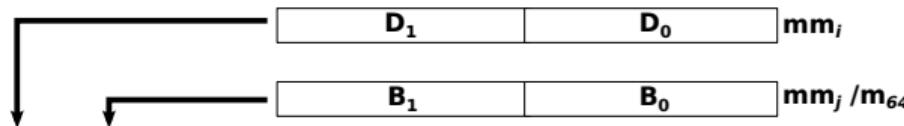
# Suma empaquetada de double words



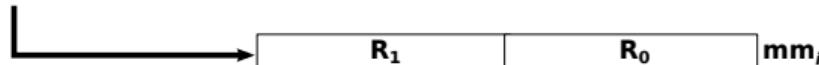
**PADDD xmm<sub>i</sub> , xmm<sub>j</sub> /m<sub>128</sub>**



$$R_k = D_k + B_k$$

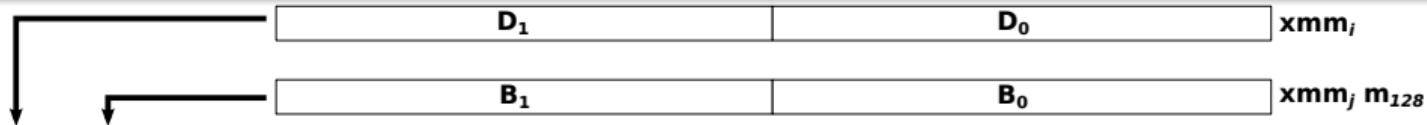


**PADDD mm<sub>i</sub> , mm<sub>j</sub> /m<sub>64</sub>**



$$R_k = D_k + B_k$$

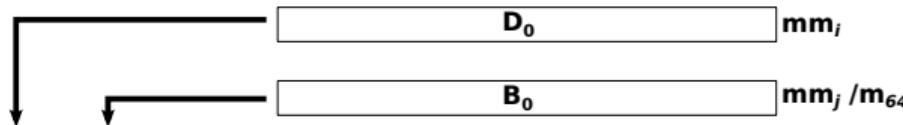
# Suma empaquetada de quad words



**PADDQ xmm<sub>i</sub> , xmm<sub>j</sub> /m<sub>128</sub>**



$$R_k = D_k + B_k$$

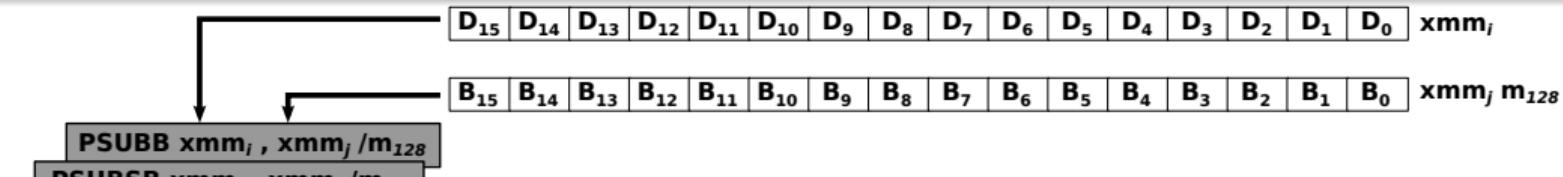


**PADDQ mm<sub>i</sub> , mm<sub>j</sub> /m<sub>64</sub>**

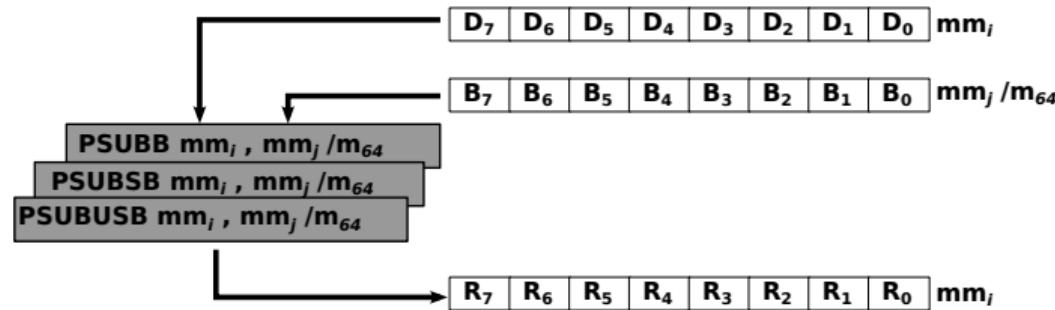


$$R_k = D_k + B_k$$

# Resta empaquetada de bytes

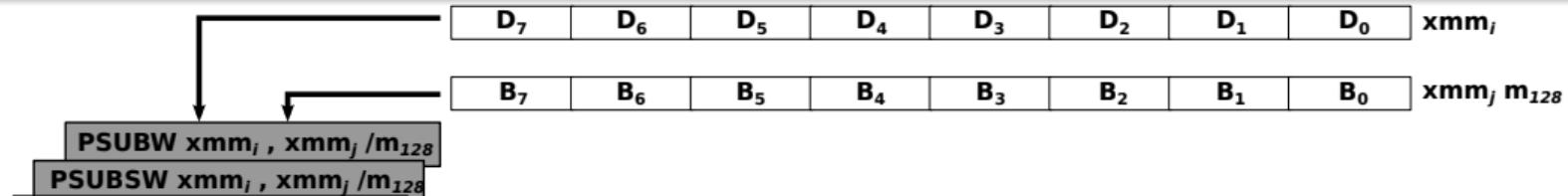


$$\mathbf{R}_k = \mathbf{D}_k - \mathbf{B}_k$$

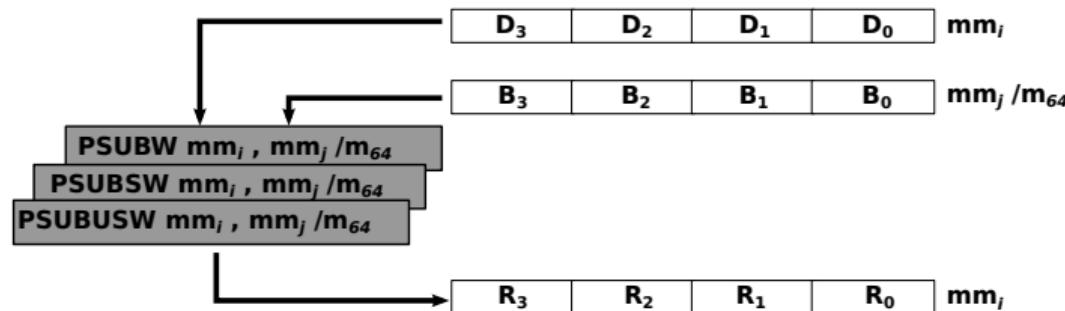


$$\mathbf{R}_k = \mathbf{D}_k - \mathbf{B}_k$$

# Resta empaquetada de words

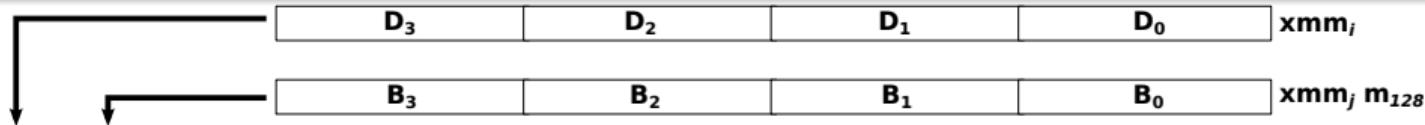


$$R_k = D_k - B_k$$

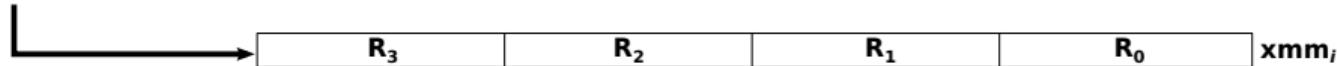


$$R_k = D_k - B_k$$

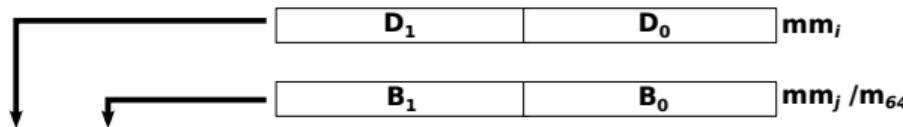
# Resta empaquetada de double words



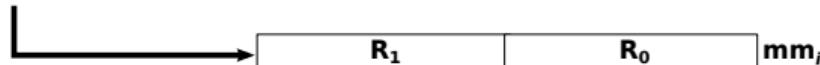
**PSUBD xmm<sub>i</sub> , xmm<sub>j</sub> /m<sub>128</sub>**



$$R_k = D_k - B_k$$

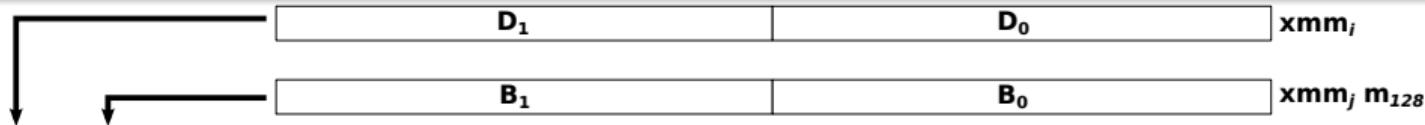


**PSUBD mm<sub>i</sub> , mm<sub>j</sub> /m<sub>64</sub>**



$$R_k = D_k - B_k$$

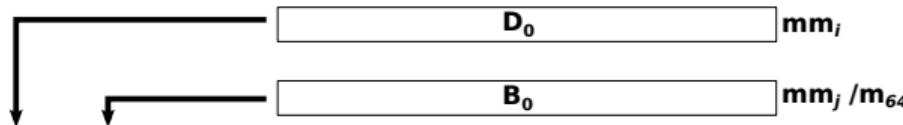
# Resta empaquetada de quad words



**PSUBQ xmm<sub>i</sub> , xmm<sub>j</sub> /m<sub>128</sub>**



$$R_k = D_k - B_k$$



**PSUBQ mm<sub>i</sub> , mm<sub>j</sub> /m<sub>64</sub>**



$$R_k = D_k - B_k$$

# 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  $n \equiv 0 \pmod{8}$ .

```
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
sumal: ; 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 |
.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
sum2: ; 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

PADD <b>SB</b>	PADD <b>SW</b>	Add Int saturation
PADD <b>USB</b>	PADD <b>USW</b>	Add Int unsigned saturation
PSUB <b>SB</b>	PSUB <b>SW</b>	Sub Int saturation
PSUB <b>USB</b>	PSUB <b>USW</b>	Sub Int unsigned saturation

# Operaciones Aritméticas

PADD <b>SB</b>	PADD <b>SW</b>	Add Int saturation
PADD <b>USB</b>	PADD <b>USW</b>	Add Int unsigned saturation
PSUB <b>SB</b>	PSUB <b>SW</b>	Sub Int saturation
PSUB <b>USB</b>	PSUB <b>USW</b>	Sub Int unsigned saturation

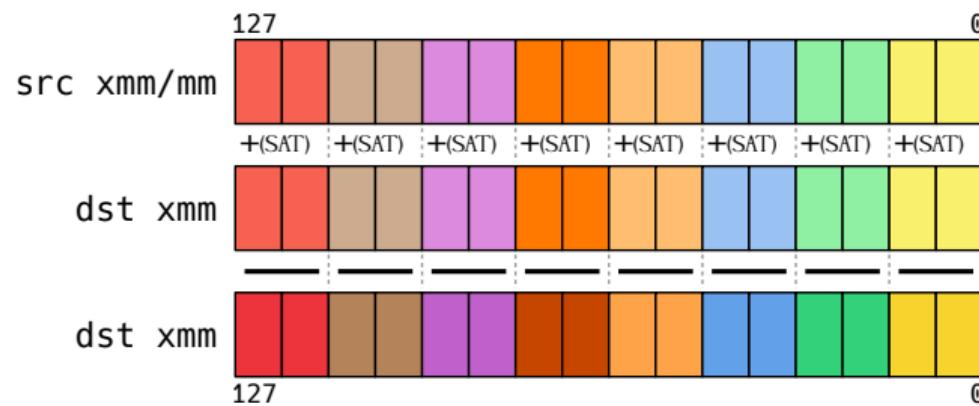
Ejemplos:

# Operaciones Aritméticas

PADD <b>SB</b>	PADD <b>SW</b>	Add Int saturation
PADD <b>USB</b>	PADD <b>USW</b>	Add Int unsigned saturation
PSUB <b>SB</b>	PSUB <b>SW</b>	Sub Int saturation
PSUB <b>USB</b>	PSUB <b>USW</b>	Sub Int unsigned saturation

Ejemplos:

PADD**SW** xmm0, xmm0

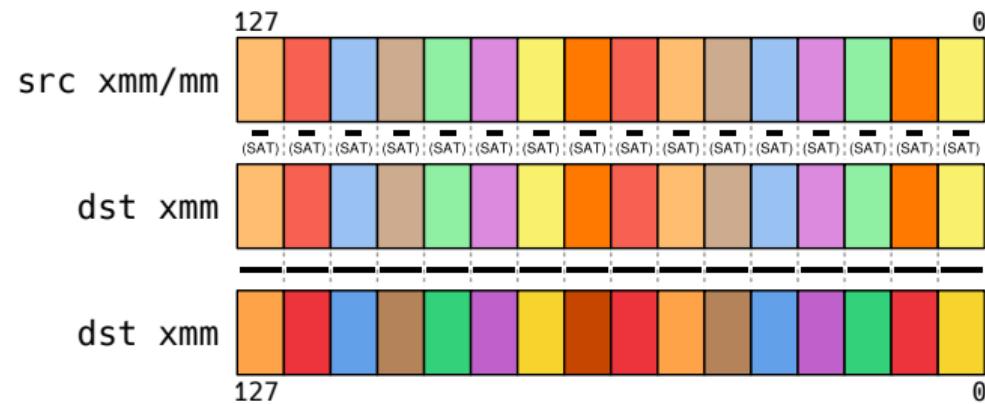


# Operaciones Aritméticas

PADD <b>SB</b>	PADD <b>SW</b>	Add Int saturation
PADD <b>USB</b>	PADD <b>USW</b>	Add Int unsigned saturation
PSUB <b>SB</b>	PSUB <b>SW</b>	Sub Int saturation
PSUB <b>USB</b>	PSUB <b>USW</b>	Sub Int unsigned saturation

Ejemplos:

PADD**SW** xmm0, xmm0 ✓  
PSUB**USB** xmm0, [data] ✓



# Operaciones Aritméticas

PADD <b>SB</b>	PADD <b>SW</b>	Add Int saturation
PADD <b>USB</b>	PADD <b>USW</b>	Add Int unsigned saturation
PSUB <b>SB</b>	PSUB <b>SW</b>	Sub Int saturation
PSUB <b>USB</b>	PSUB <b>USW</b>	Sub Int unsigned saturation

Ejemplos:

PADD**SW** xmm0, xmm0      ✓

PSUB**USB** xmm0, [data]      ✓

PSUB**SW** [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  $n \equiv 0 \pmod{8}$ .

```
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
```

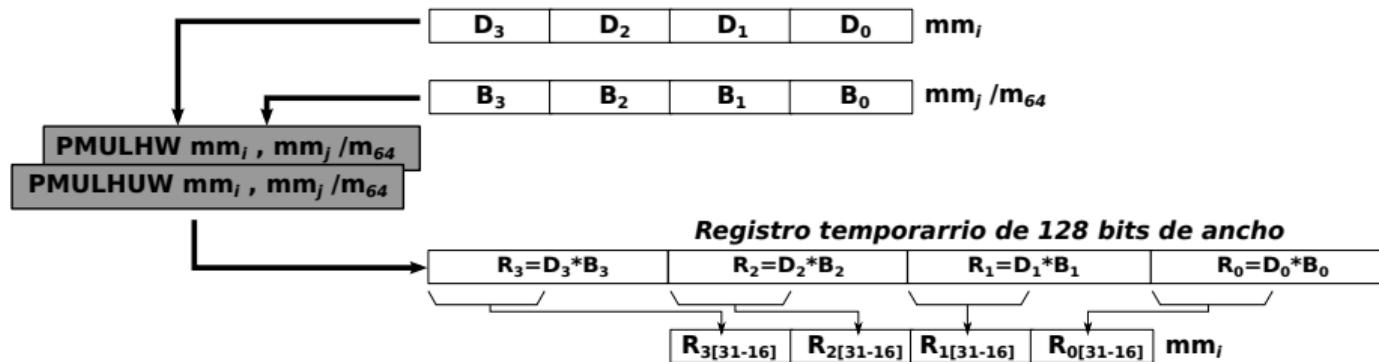
# Incrementar Brillo

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

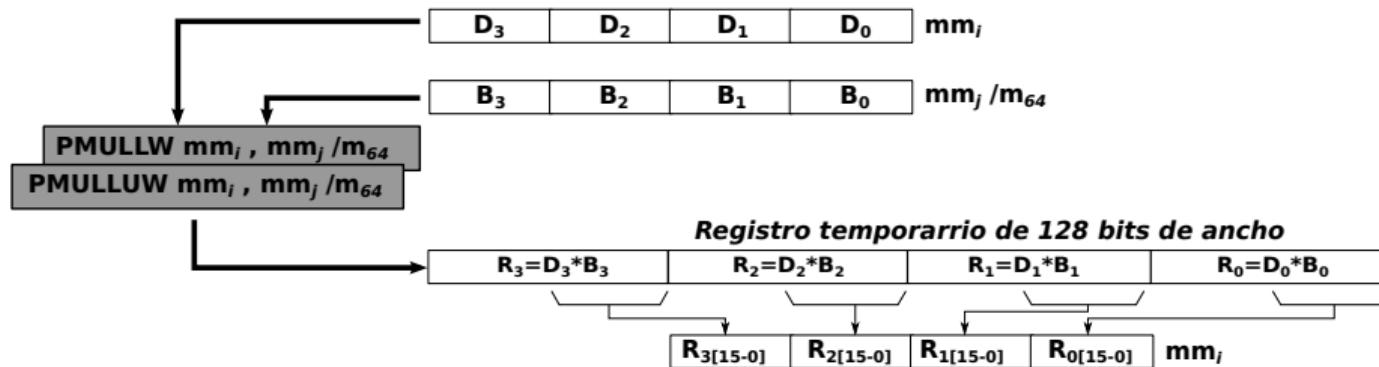
section .text

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
```

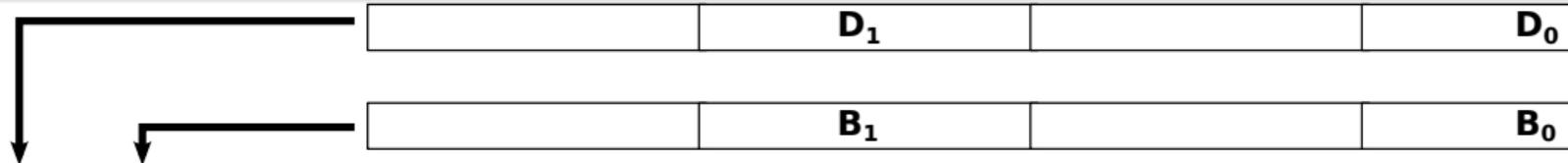
# Multiplicación de words empaquetada



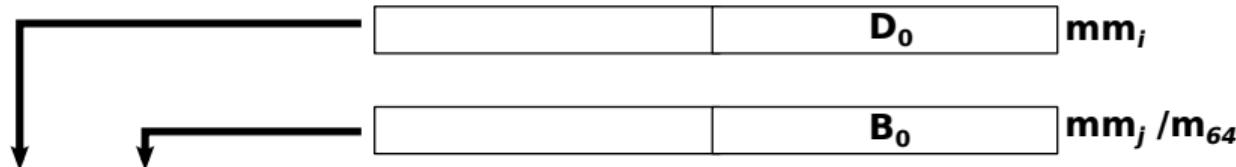
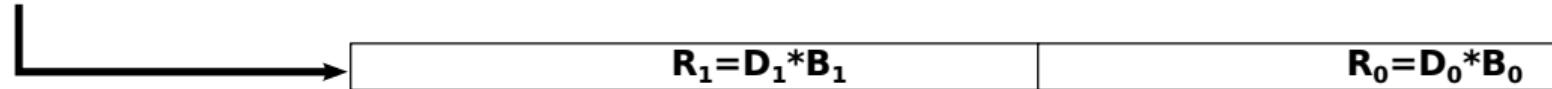
# Multiplicación de words empaquetada



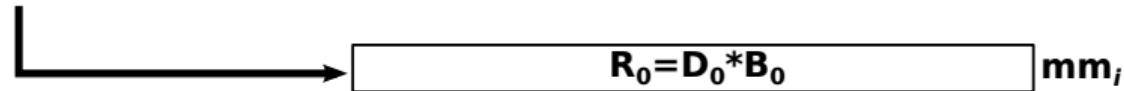
# Multiplicación de doble words empaquetada



**PMULUDQ xmm<sub>i</sub> , xmm<sub>j</sub> /m<sub>128</sub>**  
**PMULDQ xmm<sub>i</sub> , xmm<sub>j</sub> /m<sub>128</sub>**

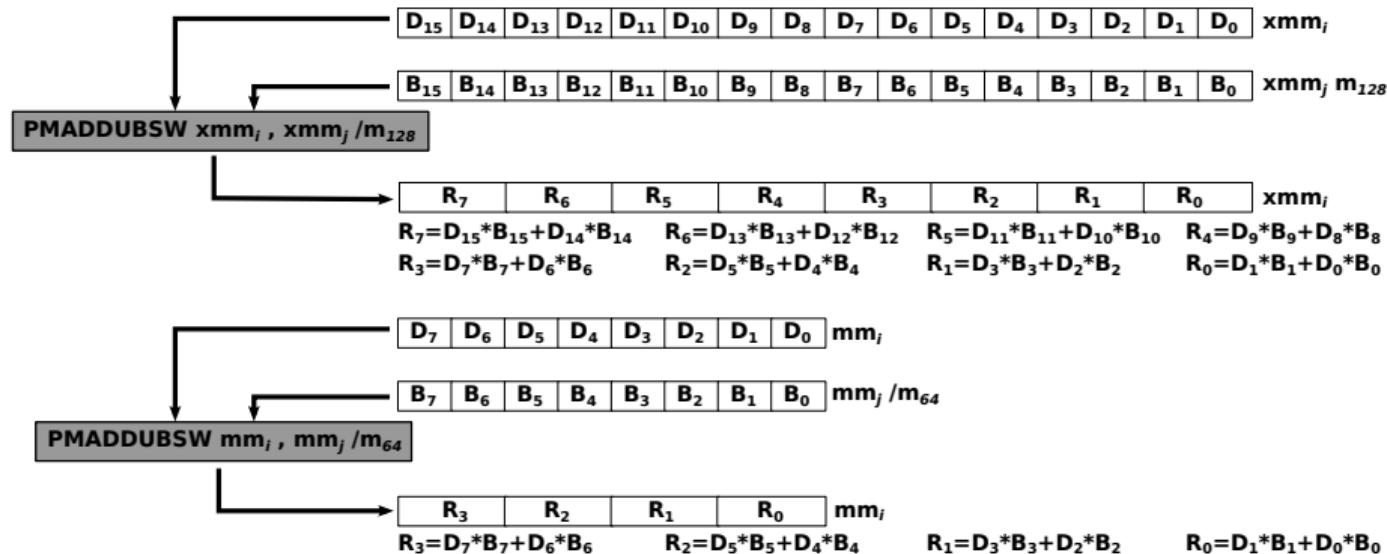


**PMULUDQ mm<sub>i</sub> , mm<sub>j</sub> /m<sub>64</sub>**



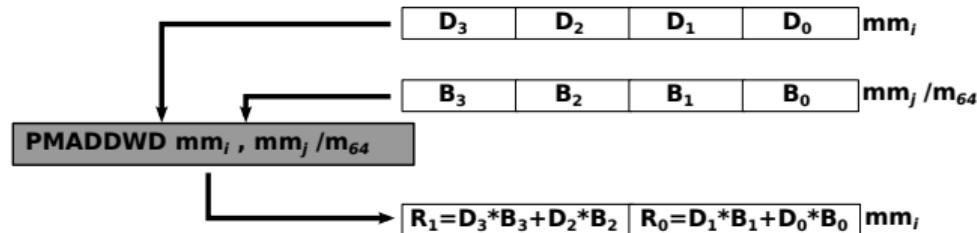
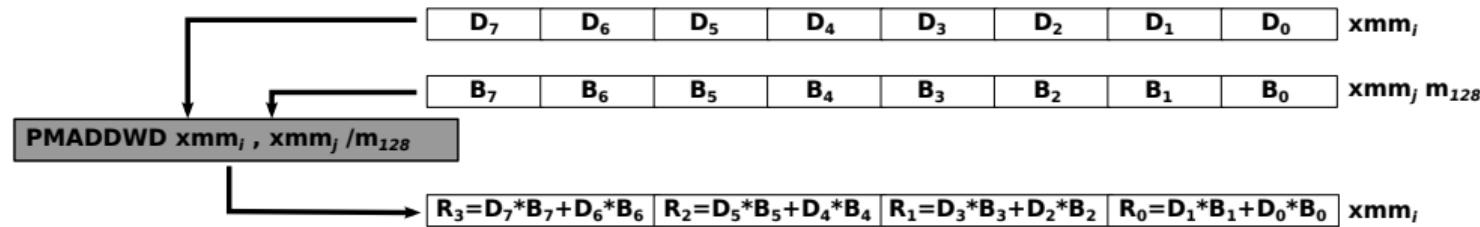
# Suma de productos empaquetada

- El primer operando contiene bytes enteros no signados
- El segundo operando contiene bytes signados empaquetados
- Almacena en el operando destino words signadas obtenidas a partir de la suma saturada de los productos parciales



# Suma de productos empaquetada

- Ambos operandos contienen words enteros signadas
- Almacena en el operando destino doble words signadas obtenidas a partir de la suma de los productos parciales



## Aplicación: Diseño de un filtro FIR

- Un filtro FIR (por **Finite Impulse Response**) son uno de los principales algoritmos de filtrado utilizados en DSP.
- La salida de un filtro de orden  $m$  viene dado por la siguiente expresión:  
$$y[j] = \sum_{i=0}^m c_i \cdot x[j - i]$$
- El siguiente fragmento de código calcula 640 valores de salida de un filtro FIR de orden 63:

```
1 for ( j = 0; j < 640; j ++ ) {  
2     int s = 0; // s = acumulador  
3     for ( i = 0; i <= 63; i ++ )  
4         s += c[i] * x[i+j]; // x[] = valores de muestras de entrada  
5                             // c[] = coeficientes del filtro  
6     y[j] = s;             // y[] = Valores de salida  
7 }
```

- $x[]$  mantiene siempre la muestra actual en la entrada mas las 639 anteriores

# Aplicación: Diseño de un filtro FIR

- Escrito con instrucciones SIMD queda del siguiente modo:

```
1 pxor xmm0, xmm0 ; inicializa en 0 4 acumuladores (c/u de 32 bits)
2 sub ebp, ebp ; inicializa Indice al buffer de coeficientes
3 Lazo2 :
4 movups xmm1, qword ptr [esi+ebp] ; carga vector de 8 elementos en registro de 128 Bits
5 pmaddwd xmm1, qword ptr [edi+ebp] ; multiplica de a pares muestras y coeficientes
6 paddd xmm0, xmm1 ; acumula resultados en xmm0 (4 x 32 Bits)
7 add ebp, 2*8 ; 2 = tamaño del dato en bytes, 8 = cantidad de datos
8 cmp ebp, 2*64 ; repite para los 64 coeficientes (tomados de a 8)
9 jnz Lazo2
10 phaddd xmm0, xmm0
11 phaddd xmm0, xmm0 ; acumula cuatro valores parciales en un resultado
12 psrad xmm0, 15 ; Escala a 16 Bits (tamaño original del dato)
13 movd qword ptr [eax], xmm0 ; Almacena (solo quedan los 16 LSB)
14 add esi, 2 ; incrementa el puntero al buffer de muestras
15 add eax, 2 ; incrementa puntero al buffer de salida
16 sub ecx, 1 ; repeat for all input samples
```

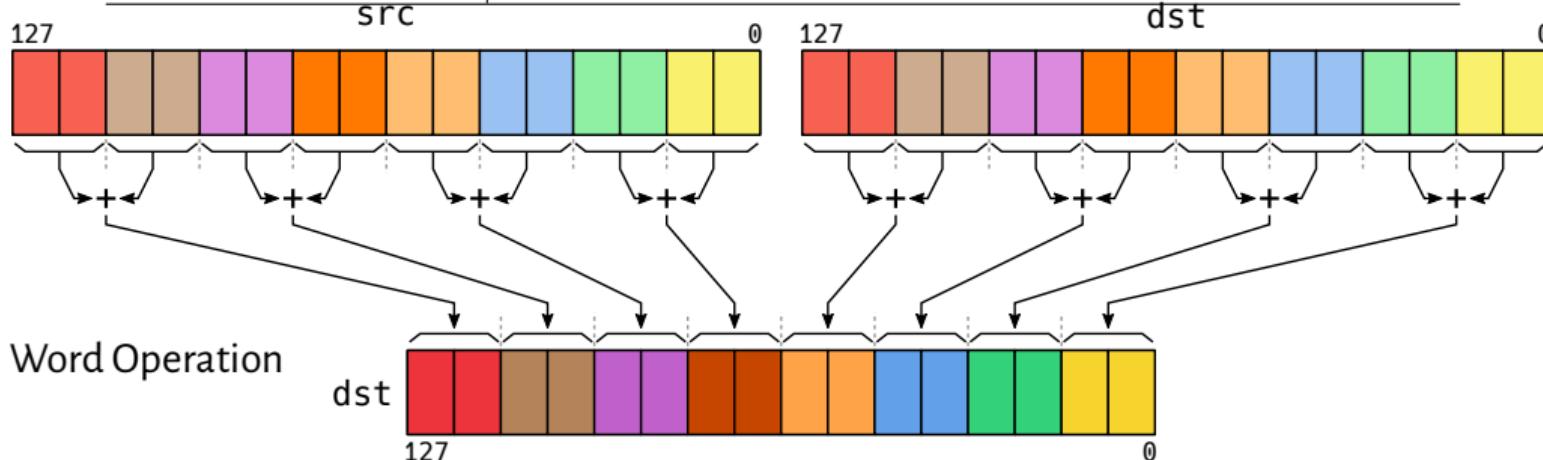
- La relación de aceleración de este algoritmo respecto del C visto anteriormente es aproximadamente 7x.

# 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
HADDP S	HADDPD	Packed Single/Double FP Horizontal Add
HSUBPS	HSUBPD	Packed Single/Double FP Horizontal Subtract

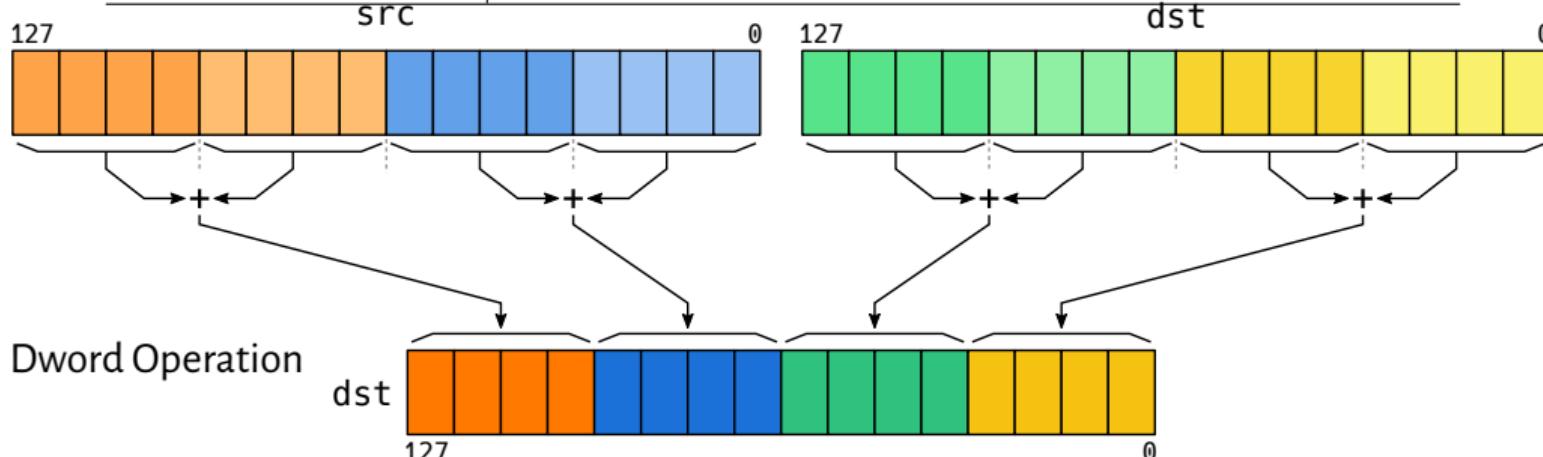
# Operaciones Aritméticas

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



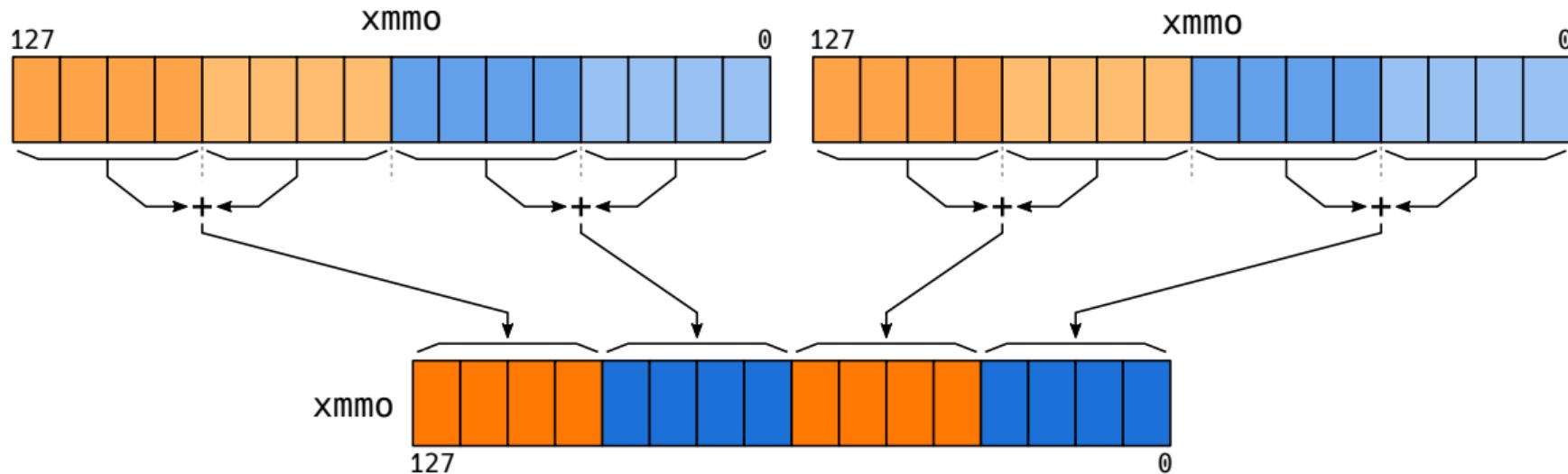
# Operaciones Aritméticas

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



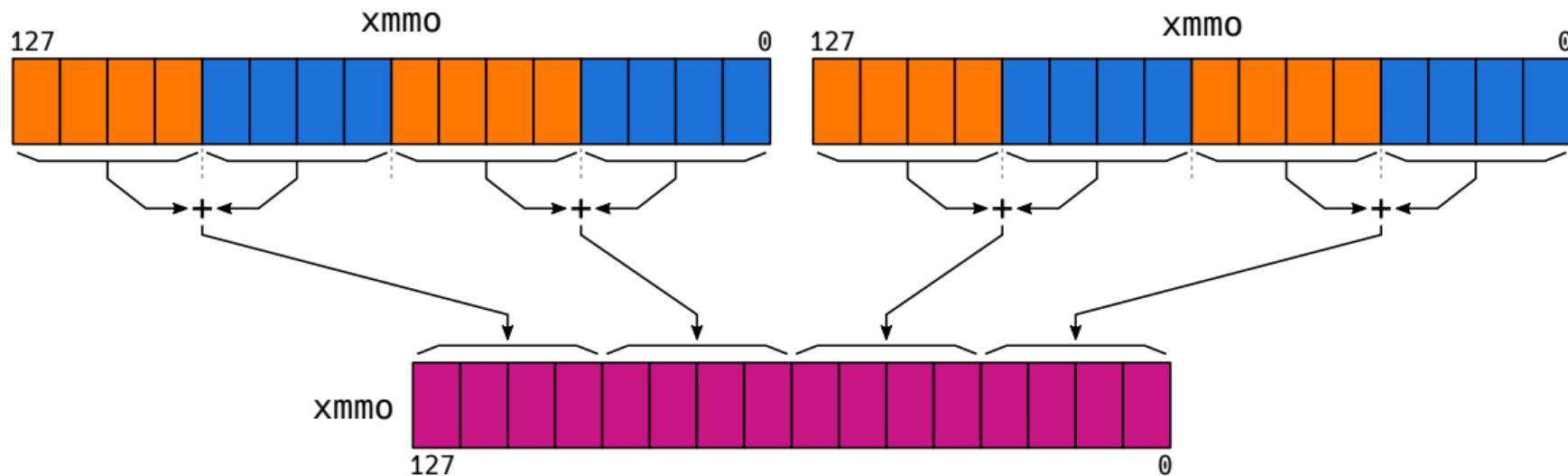
# Como funciona la suma horizontal del código anterior

- En el caso del algoritmo se la usa dos veces consecutivas y utilizando el mismo registro en ambos operandos, con el siguiente resultado:



# Como funciona la suma horizontal del código anterior

- En el caso del algoritmo se la usa dos veces consecutivas y utilizando el mismo registro en ambos operandos, con el siguiente resultado:



# Consideraciones de implementación

# Consideraciones de implementación

- En general los filtros trabajan con valores fraccionarios para los coeficientes.

# Consideraciones de implementación

- En general los filtros trabajan con valores fraccionarios para los coeficientes.
- Sin embargo como su rango dinámico nos es muy grande por lo cual considerando datos de 16 bits puede trabajarse en punto fijo cuyo resultado matemáticamente es igual que trabajar con enteros afectados por un factor de escala definido por la posición del punto decimal dentro de los 16 bits. Es decir, trabajamos con operaciones de aritmética entera pero tratamos los datos como fraccionarios de punto fijo.

# Consideraciones de implementación

- En general los filtros trabajan con valores fraccionarios para los coeficientes.
- Sin embargo como su rango dinámico nos es muy grande por lo cual considerando datos de 16 bits puede trabajarse en punto fijo cuyo resultado matemáticamente es igual que trabajar con enteros afectados por un factor de escala definido por la posición del punto decimal dentro de los 16 bits. Es decir, trabajamos con operaciones de aritmética entera pero tratamos los datos como fraccionarios de punto fijo.
- Como el rango dinámico de los coeficientes es numéricamente pequeños de modo que es de esperar que el resultado a enviar a la salida tenga una valor dentro del rango de la entrada. Por eso cada valor de salida se puede escalar a 16 bits al final de su cálculo.

# Operaciones Lógicas

PAND	PANDN	POR	PXOR	Operaciones lógicas para enteros.
ANDPS	ANDNPS	ORPS	XORPS	Operaciones lógicas para <i>float</i> .
ANDPD	ANDNPD	ORPD	XORPD	Operaciones lógicas para <i>double</i> .

- 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

11111111	00000000	11111111	00000000
----------	----------	----------	----------

1. Calculo de la mascara

# Técnica: Operatoria con mascaras

11111111|00000000|11111111|00000000

1. Calculo de la mascara

Datos



# Técnica: Operatoria con máscaras

11111111|00000000|11111111|00000000

1. Calculo de la máscara

NOT(11111111|00000000|11111111|00000000)

AND

Datos



11111111|00000000|11111111|00000000

AND



# Técnica: Operatoria con máscaras

[11111111|00000000|11111111|00000000]

1. Calculo de la máscara

NOT ([11111111|00000000|11111111|00000000])

AND

Datos

[Blue|Blue|Blue|Blue]

[00000000|Blue|00000000|Blue]

[11111111|00000000|11111111|00000000]

AND

[Green|Green|Green|Green]

[Green|00000000|Green|00000000]

2. Aplicación de la máscara

# Técnica: Operatoria con máscaras

[11111111|00000000|11111111|00000000]

1. Calculo de la máscara

NOT ([11111111|00000000|11111111|00000000])

AND

Datos



[11111111|00000000|11111111|00000000]

AND



[00000000|00000000|00000000|00000000]

2. Aplicación de la máscara

[00000000|00000000|00000000|00000000]

OR

[00000000|00000000|00000000|00000000]

# Técnica: Operatoria con máscaras

11111111|00000000|11111111|00000000

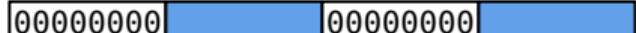
1. Calculo de la máscara

NOT ( 11111111|00000000|11111111|00000000 )

AND

Datos

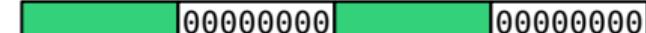




11111111|00000000|11111111|00000000

AND





2. Aplicación de la máscara

00000000|00000000|00000000|00000000

OR







3. Combinación de resultados

# 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

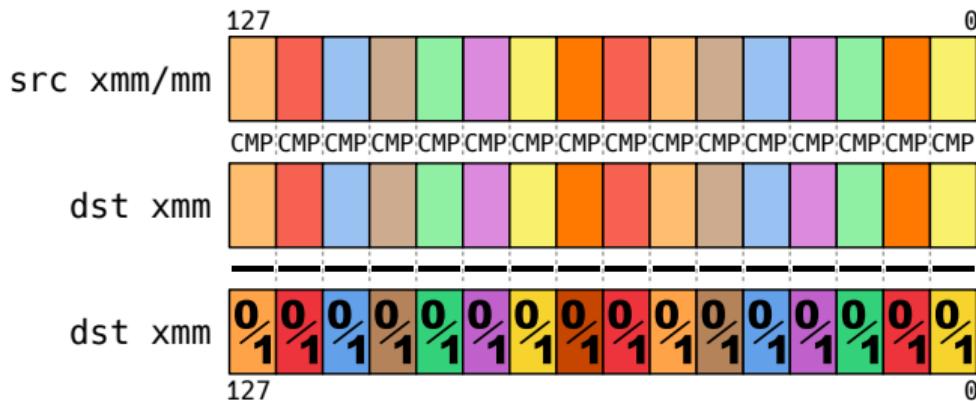
Ejemplos:

# 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] ✓

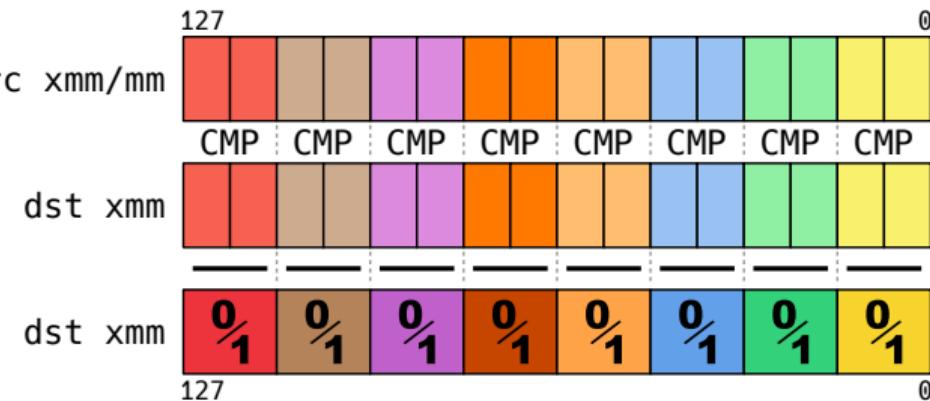


# Operaciones de comparación

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] ✓

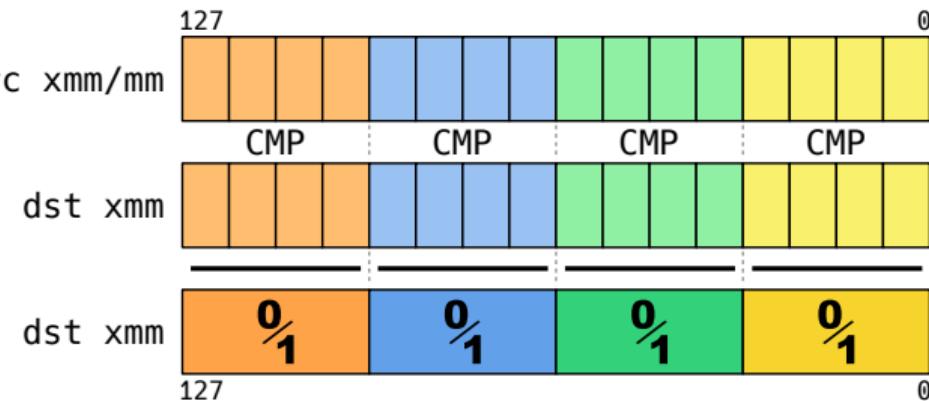


# Operaciones de comparación

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] ✓

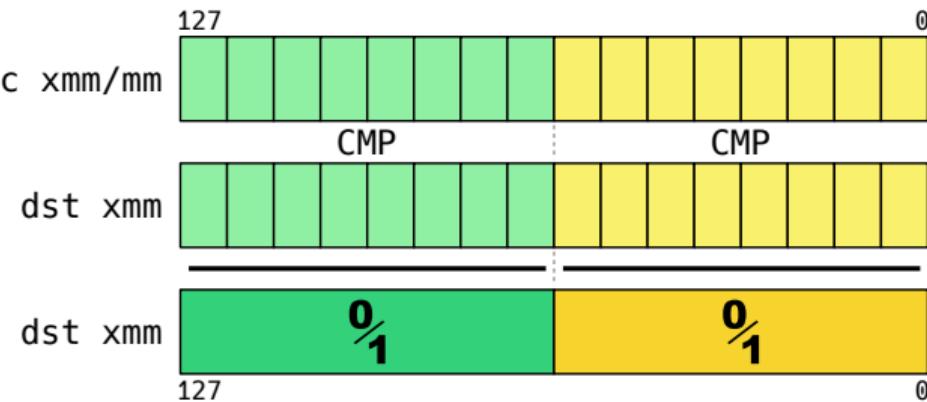


# Operaciones de comparación

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

CMP <sub>xx</sub> <b>PD</b>	Compare Packed Double-Precision Floating-Point Values
CMP <sub>xx</sub> <b>PS</b>	Compare Packed Single-Precision Floating-Point Values
CMP <sub>xx</sub> <b>SD</b>	Compare Scalar Double-Precision Floating-Point Values
CMP <sub>xx</sub> <b>SS</b>	Compare Scalar Single-Precision Floating-Point Values
COMI <b>SD</b>	Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS
COMI <b>SS</b>	Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

	Acción	<b>xx</b>	CMP <sub>xx</sub> <b>yy</b> A, B
0	Igual	EQ	$A = B$
1	Menor	LT	$A < B$
2	Menor o Igual	LE	$A \leqslant B$
3	No Orden	UNORD	$A, B = \text{unordered}$
4	Distinto	NEQ	$A \neq B$
5	No Menor	NLT	$\text{not}(A < B)$
6	No Menor o Igual	NLE	$\text{not}(A \leqslant B)$
7	Orden	ORD	$A, B = \text{Ordered}$

# Operaciones de comparación

CMP <sub>xx</sub> <b>PD</b>	Compare Packed Double-Precision Floating-Point Values
CMP <sub>xx</sub> <b>PS</b>	Compare Packed Single-Precision Floating-Point Values
CMP <sub>xx</sub> <b>SD</b>	Compare Scalar Double-Precision Floating-Point Values
CMP <sub>xx</sub> <b>SS</b>	Compare Scalar Single-Precision Floating-Point Values
COMI <b>SD</b>	Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS
COMI <b>SS</b>	Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

Ejemplos:

	Acción	<b>xx</b>	CMP <sub>xx</sub> <b>yy</b> A, B
0	Igual	EQ	$A = B$
1	Menor	LT	$A < B$
2	Menor o Igual	LE	$A \leqslant B$
3	No Orden	UNORD	$A, B = \text{unordered}$
4	Distinto	NEQ	$A \neq B$
5	No Menor	NLT	$\text{not}(A < B)$
6	No Menor o Igual	NLE	$\text{not}(A \leqslant B)$
7	Orden	ORD	$A, B = \text{Ordered}$

# Operaciones de comparación

---

CMP <sub>xx</sub> <b>PD</b>	Compare Packed Double-Precision Floating-Point Values
CMP <sub>xx</sub> <b>PS</b>	Compare Packed Single-Precision Floating-Point Values
CMP <sub>xx</sub> <b>SD</b>	Compare Scalar Double-Precision Floating-Point Values
CMP <sub>xx</sub> <b>SS</b>	Compare Scalar Single-Precision Floating-Point Values
COMI <b>SD</b>	Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS
COMI <b>SS</b>	Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

---

Ejemplos:

CMPEQPD xmm0, [data] ✓

	Acción	<b>xx</b>	CMP <sub>xx</sub> <b>yy</b> A, B
0	Igual	EQ	$A = B$
1	Menor	LT	$A < B$
2	Menor o Igual	LE	$A \leq B$
3	No Orden	UNORD	$A, B = \text{unordered}$
4	Distinto	NEQ	$A \neq B$
5	No Menor	NLT	$\text{not}(A < B)$
6	No Menor o Igual	NLE	$\text{not}(A \leq B)$
7	Orden	ORD	$A, B = \text{Ordered}$

# Operaciones de comparación

CMP <sub>xx</sub> <b>PD</b>	Compare Packed Double-Precision Floating-Point Values
CMP <sub>xx</sub> <b>PS</b>	Compare Packed Single-Precision Floating-Point Values
CMP <sub>xx</sub> <b>SD</b>	Compare Scalar Double-Precision Floating-Point Values
CMP <sub>xx</sub> <b>SS</b>	Compare Scalar Single-Precision Floating-Point Values
COMI <b>SD</b>	Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS
COMI <b>SS</b>	Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

Ejemplos:

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

	Acción	xx	CMP <sub>xx</sub> y A, B
0	Igual	EQ	$A = B$
1	Menor	LT	$A < B$
2	Menor o Igual	LE	$A \leq B$
3	No Orden	UNORD	$A, B = \text{unordered}$
4	Distinto	NEQ	$A \neq B$
5	No Menor	NLT	$\text{not}(A < B)$
6	No Menor o Igual	NLE	$\text{not}(A \leq B)$
7	Orden	ORD	$A, B = \text{Ordered}$

# Operaciones de comparación

CMP <sub>xx</sub> <b>PD</b>	Compare Packed Double-Precision Floating-Point Values
CMP <sub>xx</sub> <b>PS</b>	Compare Packed Single-Precision Floating-Point Values
CMP <sub>xx</sub> <b>SD</b>	Compare Scalar Double-Precision Floating-Point Values
CMP <sub>xx</sub> <b>SS</b>	Compare Scalar Single-Precision Floating-Point Values
COMI <b>SD</b>	Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS
COMI <b>SS</b>	Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

Ejemplos:

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

Acción	xx	CMP <sub>xx</sub> y A, B
Igual	EQ	$A = B$
Menor	LT	$A < B$
Menor o Igual	LE	$A \leqslant B$
No Orden	UNORD	$A, B = \text{unordered}$
Distinto	NEQ	$A \neq B$
No Menor	NLT	$\text{not}(A < B)$
No Menor o Igual	NLE	$\text{not}(A \leqslant B)$
Orden	ORD	$A, B = \text{Ordered}$

# 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
```

# Instrucciones de Comparación

# Instrucciones de Comparación

- En el caso de que la condición sea EQ (Equal) cada dato empaquetado del operando destino que cumpla la condición se pone en 0xFF, 0xFFFF, o 0xFFFFFFFF, según el tipo de dato empaquetado que se evalúe.

# Instrucciones de Comparación

- En el caso de que la condición sea EQ (Equal) cada dato empaquetado del operando destino que cumpla la condición se pone en 0xFF, 0xFFFF, o 0xFFFFFFFF, según el tipo de dato empaquetado que se evalúe.
- Si la condición no se cumple se pone a 0.

# Instrucciones de Comparación

- En el caso de que la condición sea EQ (Equal) cada dato empaquetado del operando destino que cumpla la condición se pone en 0xFF, 0xFFFF, o 0xFFFFFFFF, según el tipo de dato empaquetado que se evalúe.
- Si la condición no se cumple se pone a 0.
- Son instrucciones útiles para operaciones tan simples como inicializar en 1's un determinado registro, como por ejemplo:

PCMPEQB xmm0, xmm0 ; xmm0 = FF FF.

# Instrucciones de Comparación

- En el caso de que la condición sea EQ (Equal) cada dato empaquetado del operando destino que cumpla la condición se pone en 0xFF, 0xFFFF, o 0xFFFFFFFF, según el tipo de dato empaquetado que se evalúe.
- Si la condición no se cumple se pone a 0.
- Son instrucciones útiles para operaciones tan simples como inicializar en 1's un determinado registro, como por ejemplo:

PCMPEQB xmm0, xmm0 ; xmm0 = FF FF.

- También para operaciones algo mas sofisticadas como generación de máscaras de modo de procesar bits de manera mas eficiente.

# Ejemplo de uso de comparación empaquetada

Procesamiento de sprites o iconos en gráficos 2D.

# Ejemplo de uso de comparación empaquetada

## Procesamiento de sprites o iconos en gráficos 2D.

- Un sprite es un mapa de bits en el que se dibuja una figura pequeña, de modo tal que los bits que no se utilizan deben quedar “transparentes”, es decir, quedan del color del fondo.

# Ejemplo de uso de comparación empaquetada

## Procesamiento de sprites o iconos en gráficos 2D.

- Un sprite es un mapa de bits en el que se dibuja una figura pequeña, de modo tal que los bits que no se utilizan deben quedar “transparentes”, es decir, quedan del color del fondo.
- Supongamos un sprite de 8 pixels de ancho por otros 8 pixels de alto.

# Ejemplo de uso de comparación empaquetada

## Procesamiento de sprites o iconos en gráficos 2D.

- Un sprite es un mapa de bits en el que se dibuja una figura pequeña, de modo tal que los bits que no se utilizan deben quedar “transparentes”, es decir, quedan del color del fondo.
- Supongamos un sprite de 8 pixels de ancho por otros 8 pixels de alto.
- Para la primer línea de 8 pixels, el dibujo del sprite corresponde a los primeros cuatro mientras que los cuatro siguientes deben ser transparentes, es decir, deben ir del color del fondo.

# Ejemplo de uso de comparación empaquetada

## Procesamiento de sprites o iconos en gráficos 2D.

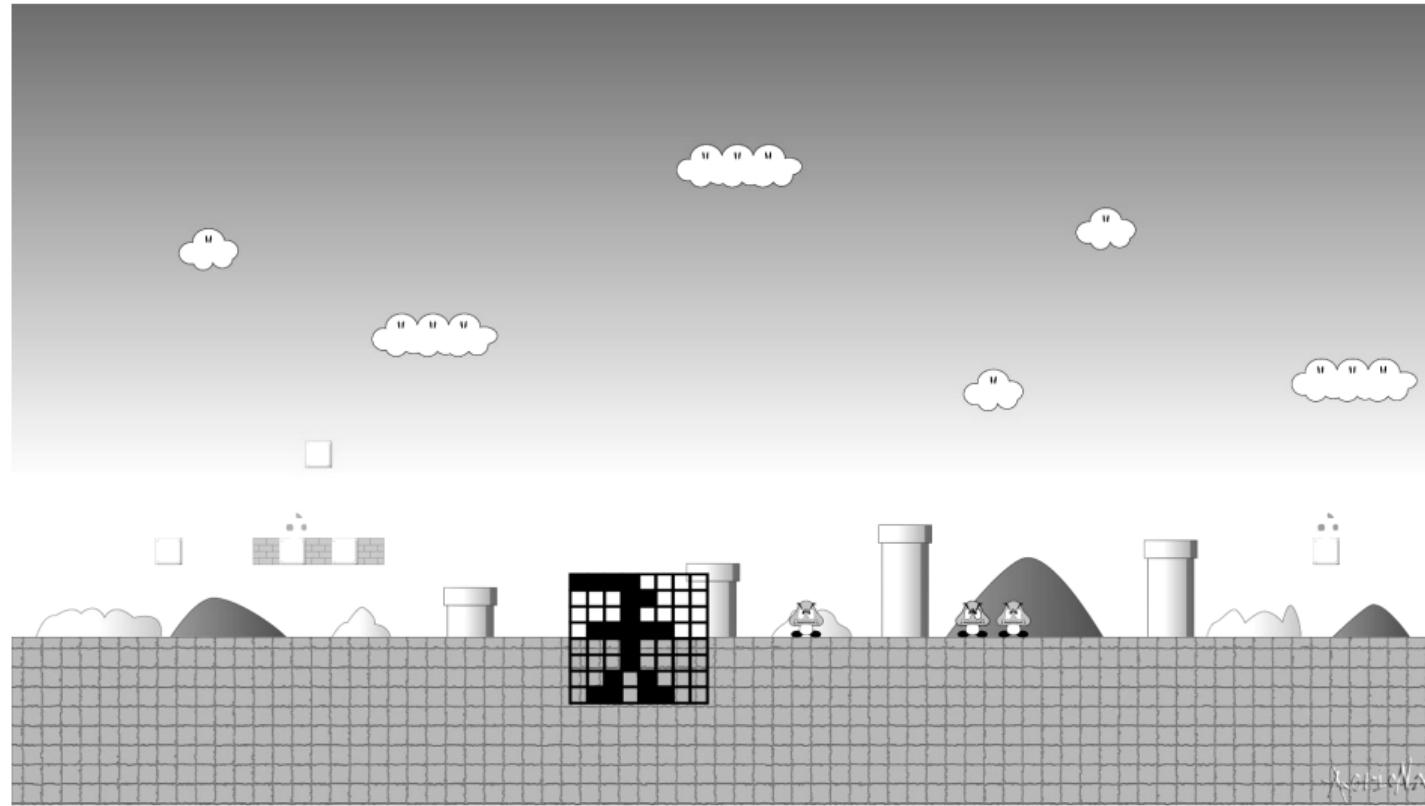
- Un sprite es un mapa de bits en el que se dibuja una figura pequeña, de modo tal que los bits que no se utilizan deben quedar “transparentes”, es decir, quedan del color del fondo.
- Supongamos un sprite de 8 pixels de ancho por otros 8 pixels de alto.
- Para la primer línea de 8 pixels, el dibujo del sprite corresponde a los primeros cuatro mientras que los cuatro siguientes deben ser transparentes, es decir, deben ir del color del fondo.
- Cargamos en mm0 la primer línea del sprite, (8 bytes empaquetados), y mm2 trabaja como auxiliar, está pre seteado en cero.

# Ejemplo de uso de comparación empaquetada

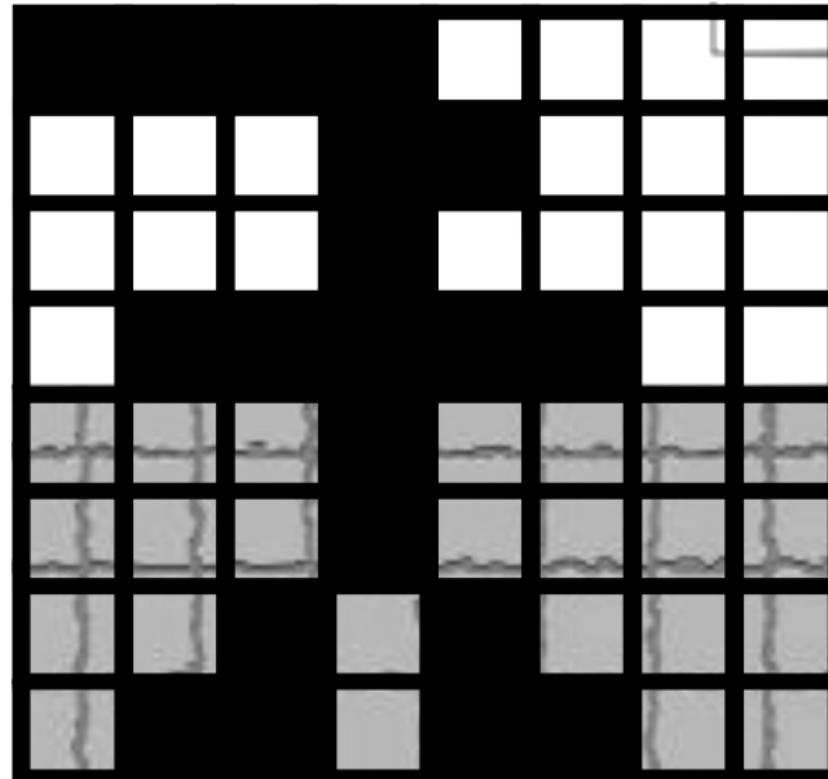
## Procesamiento de sprites o iconos en gráficos 2D.

- Un sprite es un mapa de bits en el que se dibuja una figura pequeña, de modo tal que los bits que no se utilizan deben quedar “transparentes”, es decir, quedan del color del fondo.
- Supongamos un sprite de 8 pixels de ancho por otros 8 pixels de alto.
- Para la primer línea de 8 pixels, el dibujo del sprite corresponde a los primeros cuatro mientras que los cuatro siguientes deben ser transparentes, es decir, deben ir del color del fondo.
- Cargamos en mm0 la primer línea del sprite, (8 bytes empaquetados), y mm2 trabaja como auxiliar, está pre seteado en cero.
- Luego PCMPEQB entre ambos registros, generará una máscara con los bytes que contendrán información del sprite en cero y los que deben resultar transparentes en 0xFF.

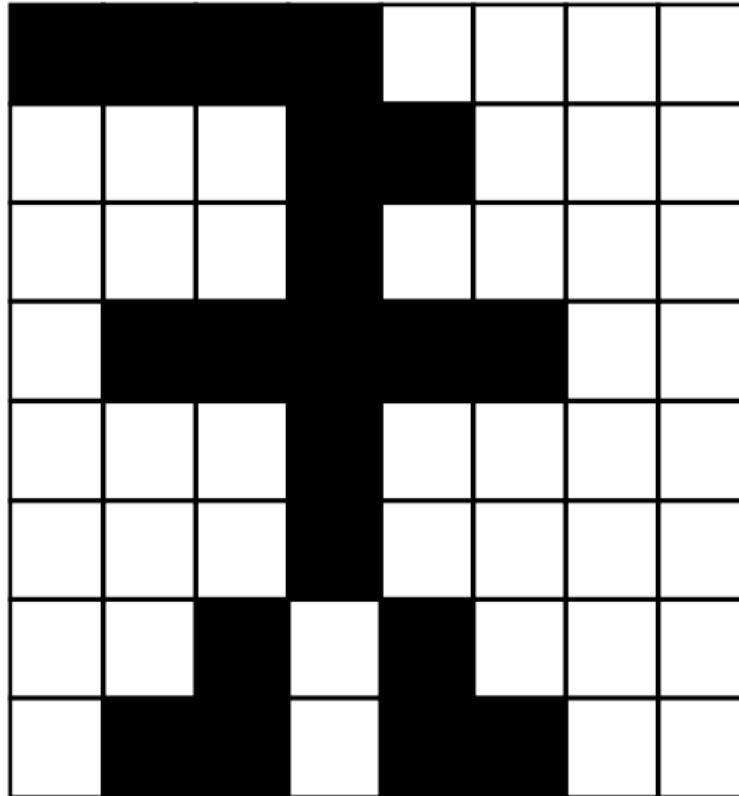
# Ejemplo de uso de comparación empaquetada



# Ejemplo de uso de comparación empaquetada



# Ejemplo de uso de comparación empaquetada



# Ejemplo de uso de comparación empaquetada

X	X	X	X	0	0	0	0
0	0	0	X	X	0	0	0
0	0	0	X	0	0	0	0
0	X	X	X	X	X	0	0
0	0	0	X	0	0	0	0
0	0	0	X	0	0	0	0
0	0	X	0	X	0	0	0
0	X	X	0	X	X	0	0

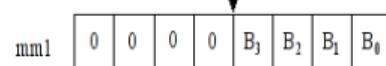
# Ejemplo de uso de comparación empaquetada



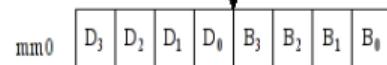
PCMPEQB mm2, mm0



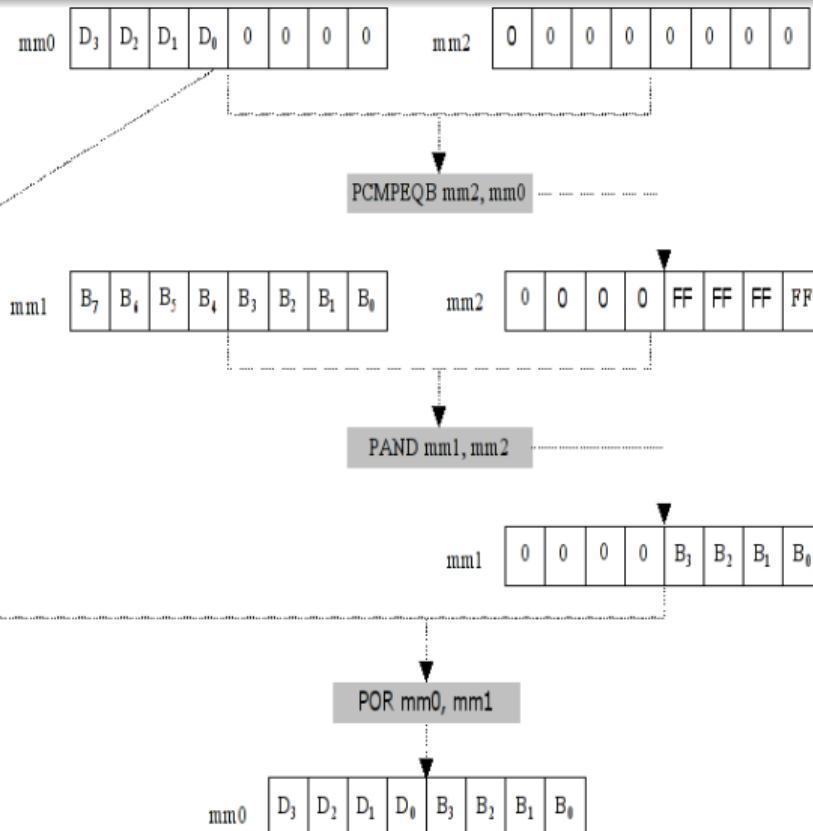
PAND mm1, mm2



POR mm0, mm1

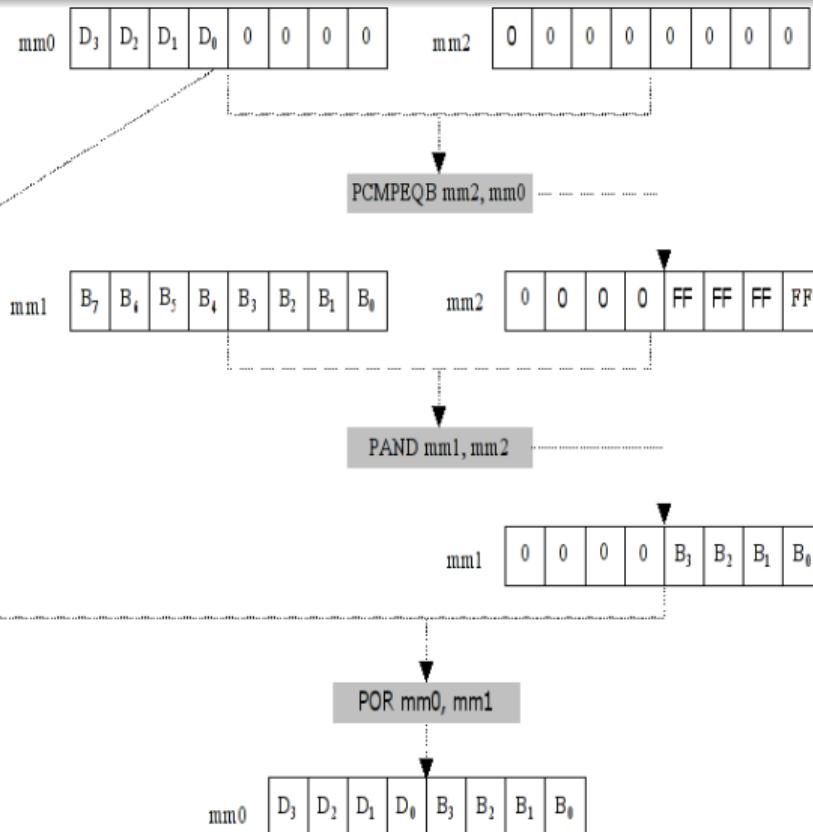


# Ejemplo de uso de comparación empaquetada



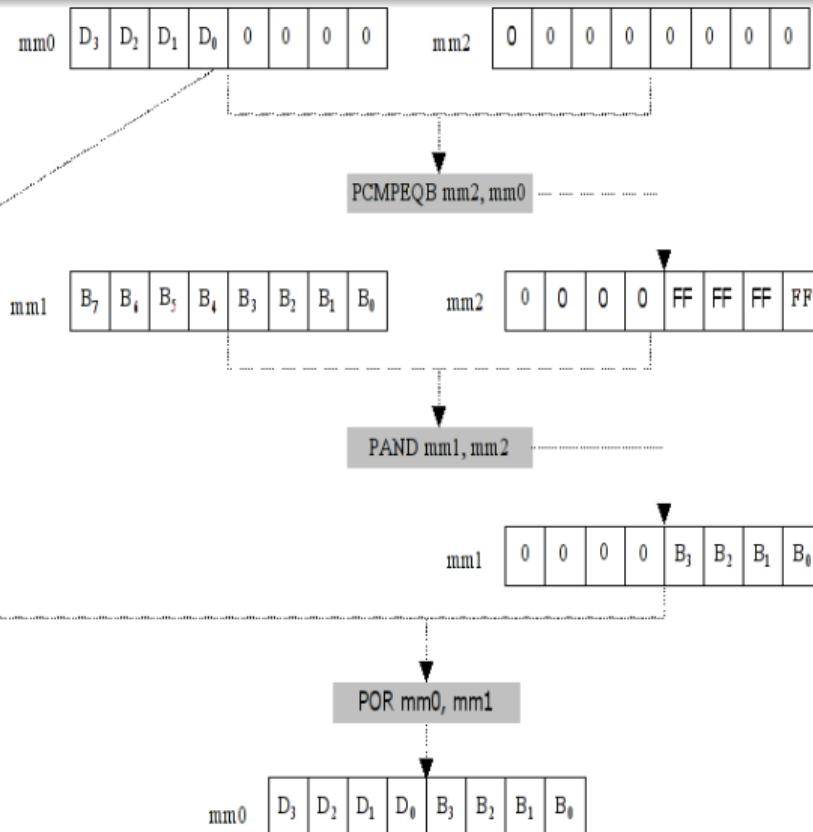
- El resultado se procesa con la información de los 8 pixels del fondo de la imagen (pre almacenado en el registro mm1) a través de un operador AND. Al resultado le queden solo los cuatro pixels del fondo que se deben mostrar.

# Ejemplo de uso de comparación empaquetada



- El resultado se procesa con la información de los 8 pixels del fondo de la imagen (pre almacenado en el registro mm1) a través de un operador AND. Al resultado le queden solo los cuatro pixels del fondo que se deben mostrar.
- Nos queda imprimir la línea del sprite sobre este fondo, del cual hemos eliminado los cuatro pixels que serán sobre impresos por la línea del sprite.

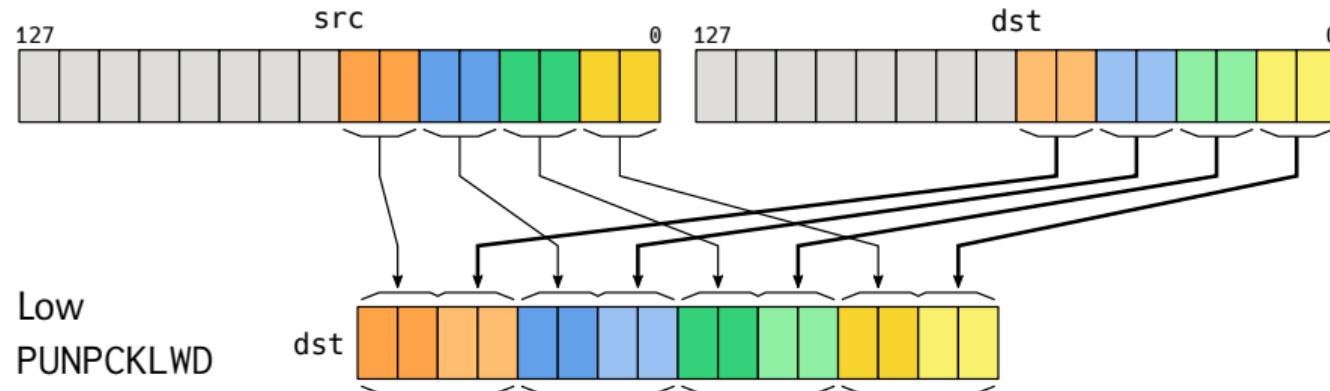
# Ejemplo de uso de comparación empaquetada



- El resultado se procesa con la información de los 8 pixels del fondo de la imagen (pre almacenado en el registro mm1) a través de un operador AND. Al resultado le queden solo los cuatro pixels del fondo que se deben mostrar.
- Nos queda imprimir la línea del sprite sobre este fondo, del cual hemos eliminado los cuatro pixels que serán sobre impresos por la línea del sprite.
- Tratando a este resultado mediante una operación OR con los 8 pixels originales del sprite se compone el dibujo buscado sobre el fondo existente.

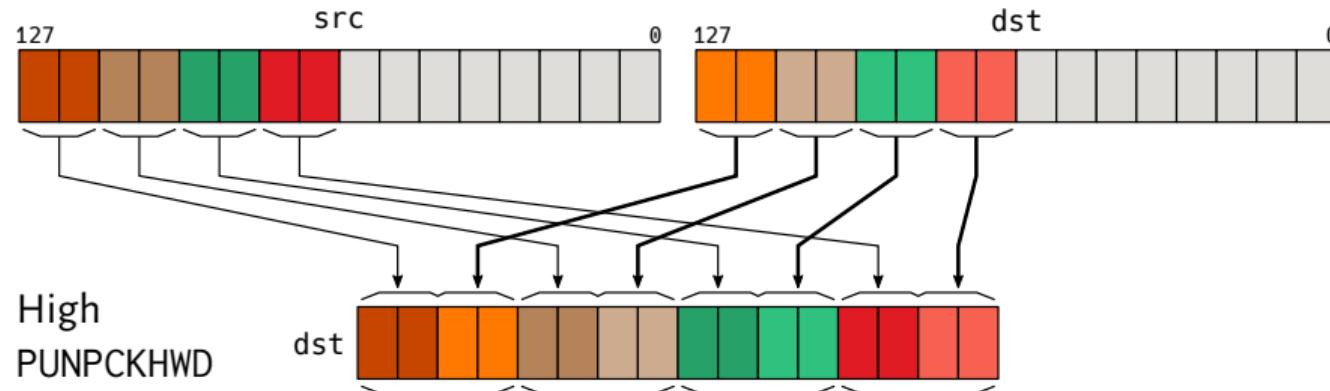
# Operaciones de desempaquetado (Unpack)

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

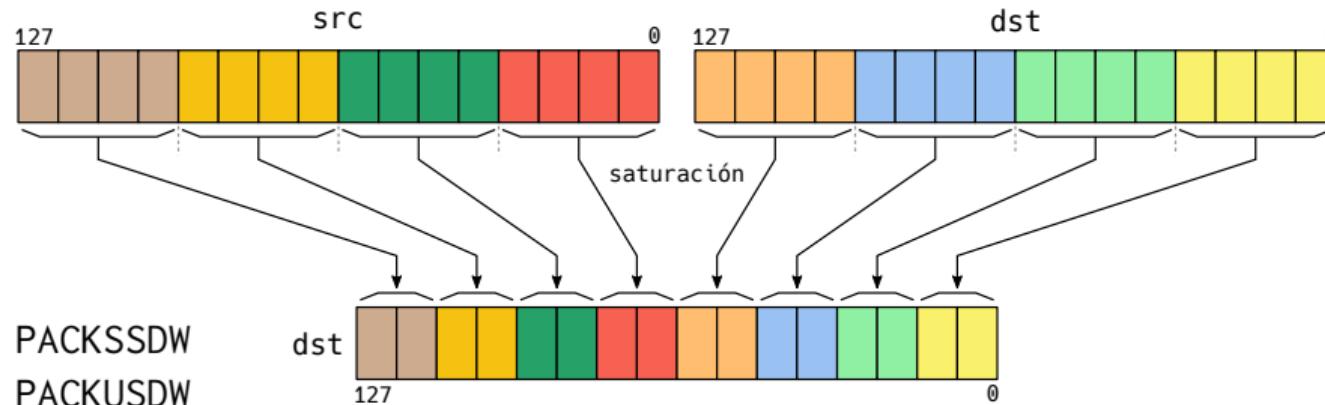


# Operaciones de desempaquetado (Unpack)

---

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

Dato

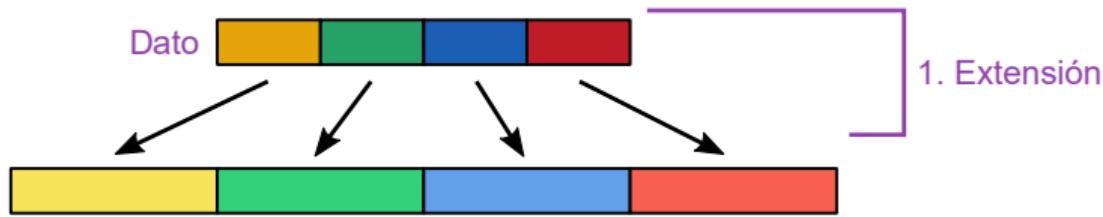


# Técnica: Operatoria de desempaquetado y empaquetado

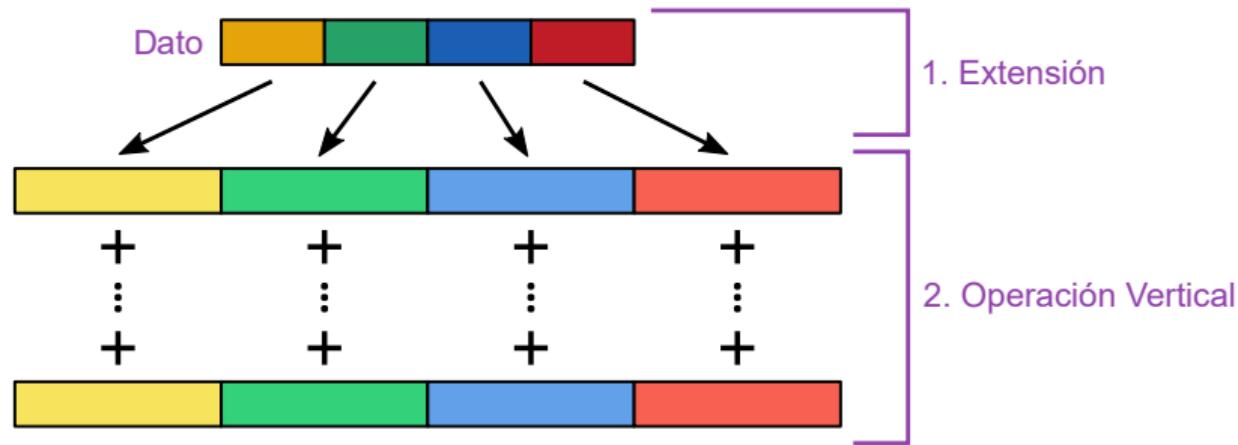
Dato



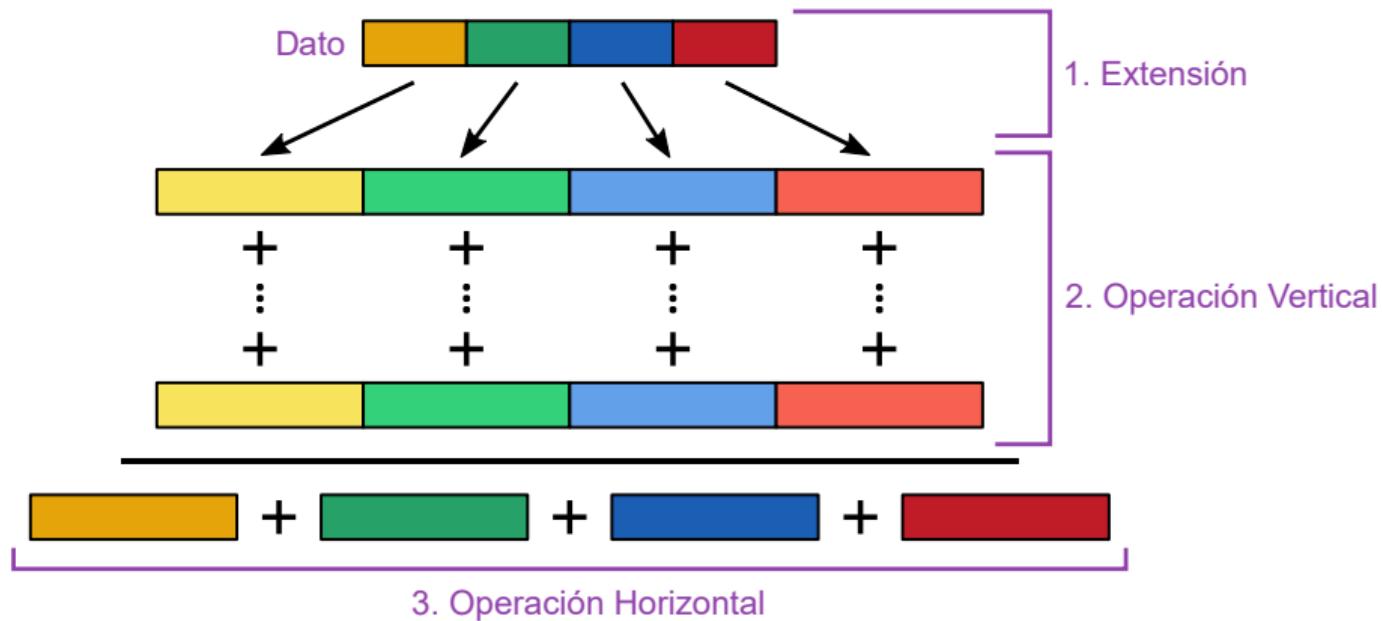
# Técnica: Operatoria de desempaquetado y empaquetado



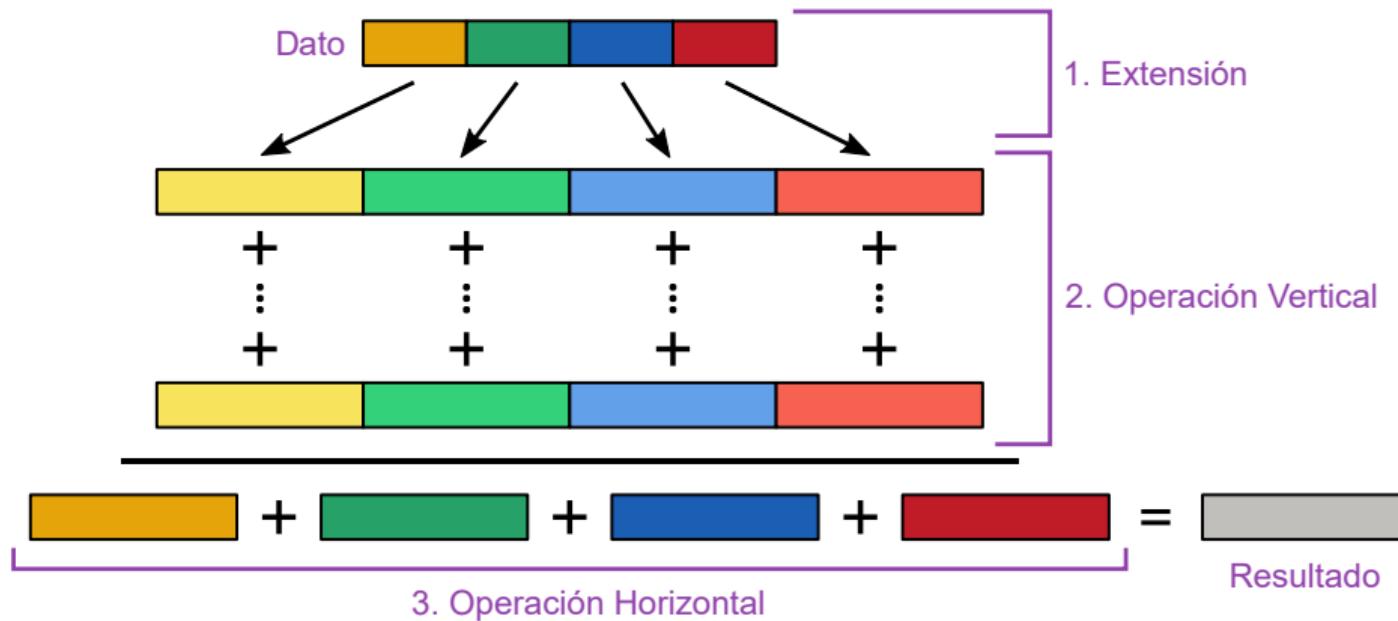
# Técnica: Operatoria de desempaquetado y empaquetado



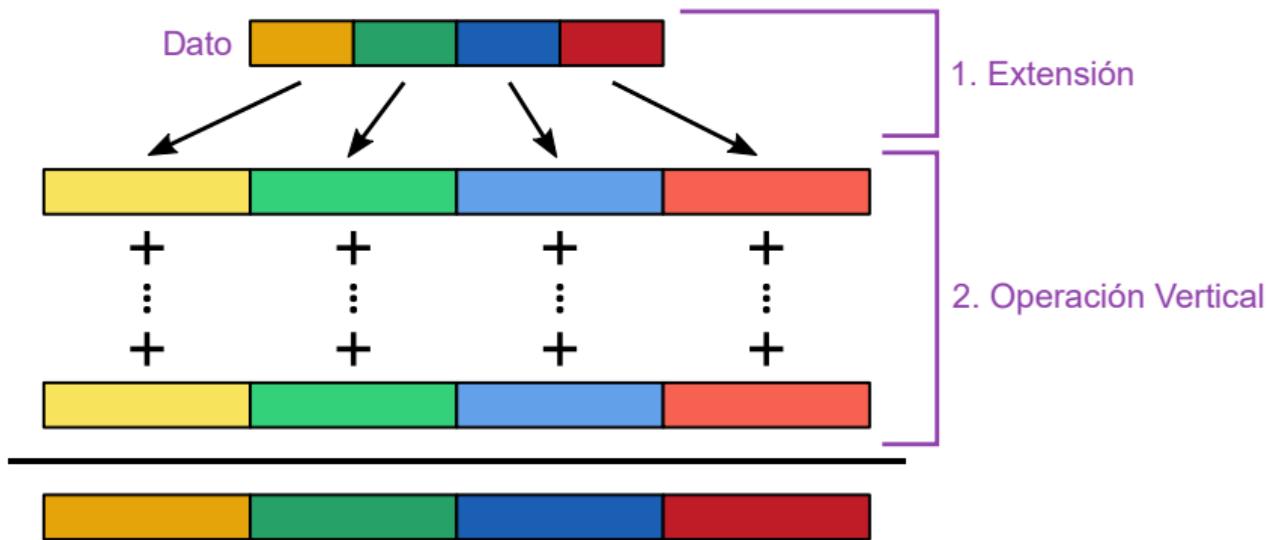
# Técnica: Operatoria de desempaquetado y empaquetado



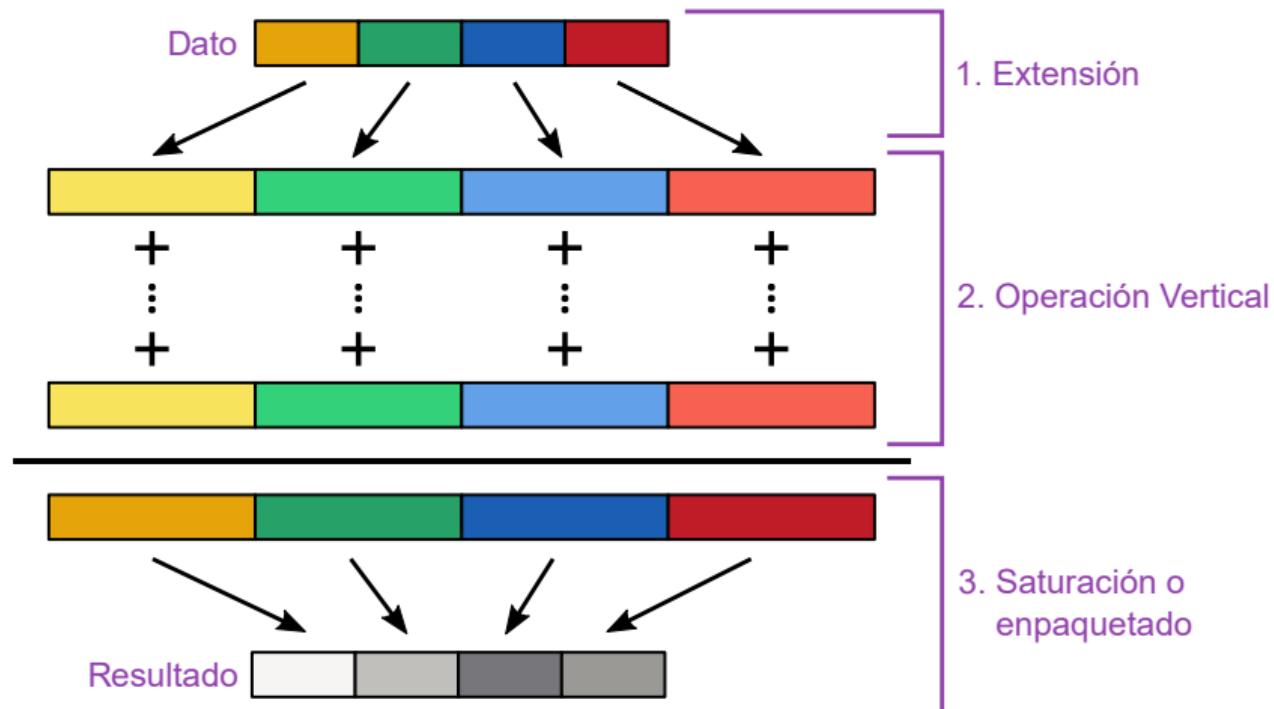
# Técnica: Operatoria de desempaquetado y empaquetado



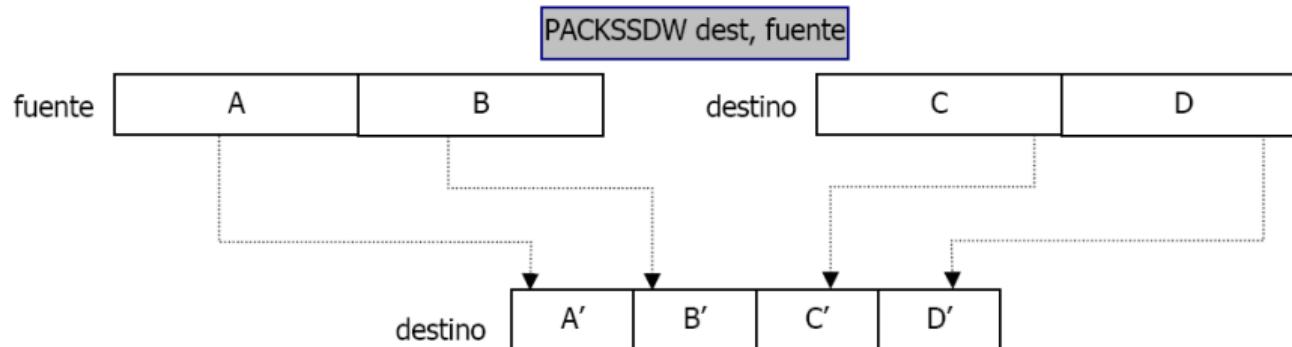
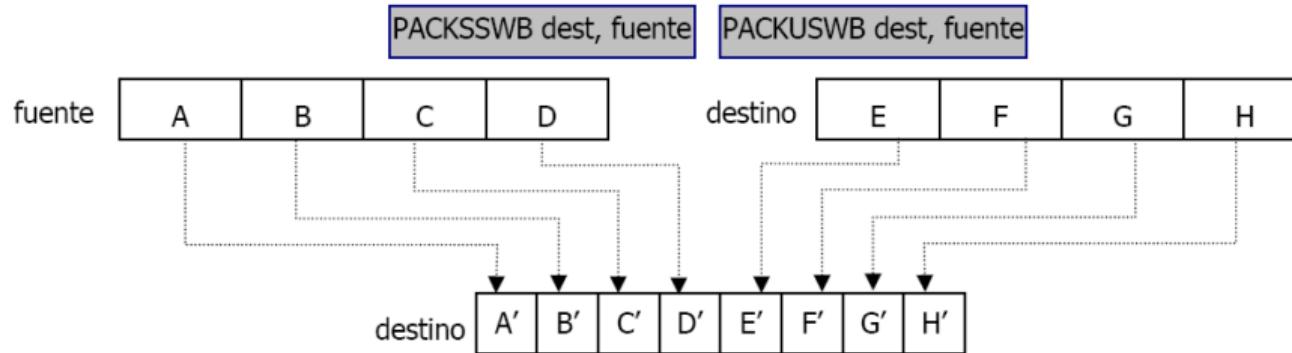
# Técnica: Operatoria de desempaquetado y empaquetado



# Técnica: Operatoria de desempaquetado y empaquetado



# Conversiones de datos empaquetados enteros



# 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

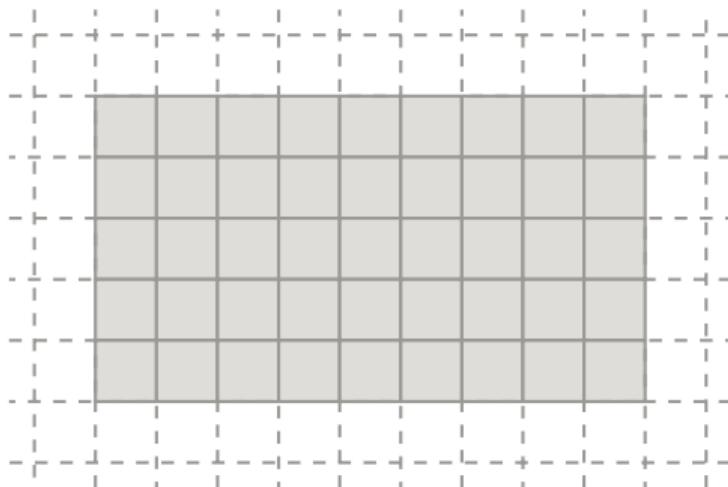
1	2	1
2	4	2
1	2	1

# Técnica: Operatoria con un kernel

$$\left( \text{datos} * \begin{array}{|ccc|} \hline & 1 & 2 & 1 \\ \hline 1 & 2 & 4 & 2 \\ \hline & 1 & 2 & 1 \\ \hline \end{array} \right) / 16$$

# Técnica: Operatoria con un kernel

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



# Técnica: Operatoria con un kernel

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

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

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

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

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

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

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

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

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

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

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

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

$$\left( \text{datos} * \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array} \right) / 16$$
$$( 1 * \boxed{0 \ B \ 8 \ 0} )$$

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

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

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

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

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

$$( 1 * \begin{array}{ccccc} 0 & B & 8 & 0 \end{array} ) + \\ 2 * \begin{array}{ccccc} 9 & 0 & 1 & 2 \end{array} + \\ 1 * \begin{array}{ccccc} 6 & 1 & 8 & 2 \end{array} +$$

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

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

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

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

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

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

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

# Técnica: Operatoria con un kernel

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

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

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

# Técnica: Operatoria con un kernel

$$\left( \text{datos} * \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array} \right) / 16$$

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

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

# Técnica: Operatoria con un kernel

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

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

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

# Técnica: Operatoria con un kernel

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

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 +
 4* 0 | 1 | 2 | 4 +
 2* 1 | 8 | 2 | F +
 1* 8 | 0 | 2 | 4 +
 2* 1 | 2 | 4 | 4 +
 1* 8 | 2 | F | 4 )
```

# Técnica: Operatoria con un kernel

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

0	1	C	D	E	7	0	2	5
A	0	B	8	0	2	4	5	D
9	9	4	3	3	4	4	0	5
6	6	1	8	2	F	4	5	2
6	0	5	F	2	1	1	6	6

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

# Ejemplo

## Efecto Blur

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

```
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
```

# Ejemplo de procesamiento de audio

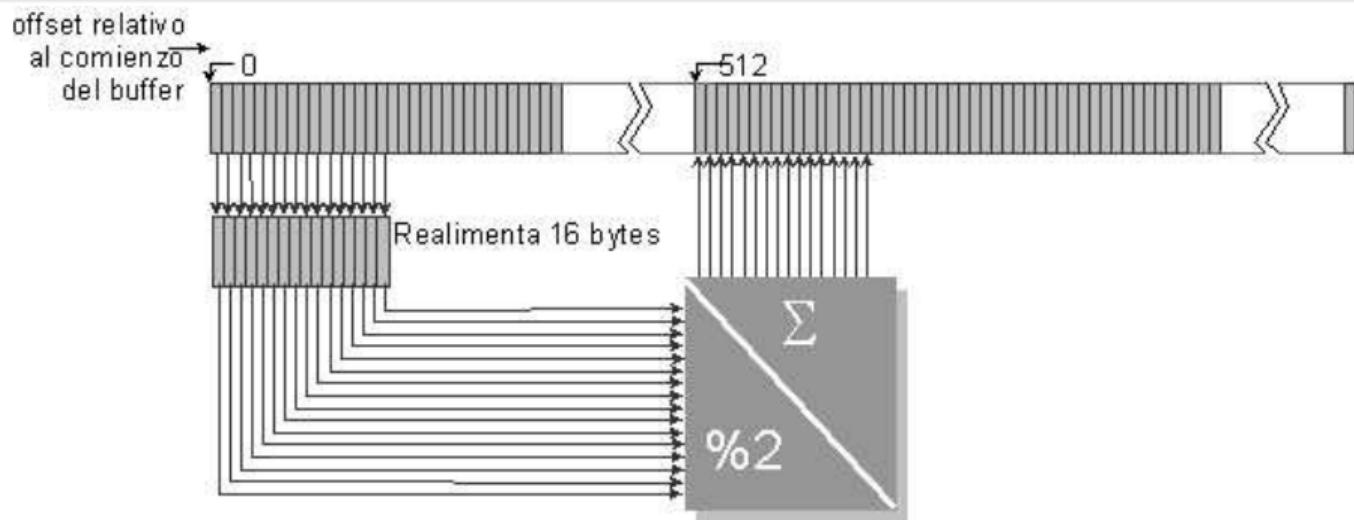
## Eco simple

El algoritmo está pensado con de modo tal de comenzar a realimentar las muestras hacia la entrada con una demora equivalente al tiempo de 512 muestras de modo que el efecto resulte apreciable.

# Ejemplo de procesamiento de audio

## Eco simple

El algoritmo está pensado con de modo tal de comenzar a realimentar las muestras hacia la entrada con una demora equivalente al tiempo de 512 muestras de modo que el efecto resulte apreciable.



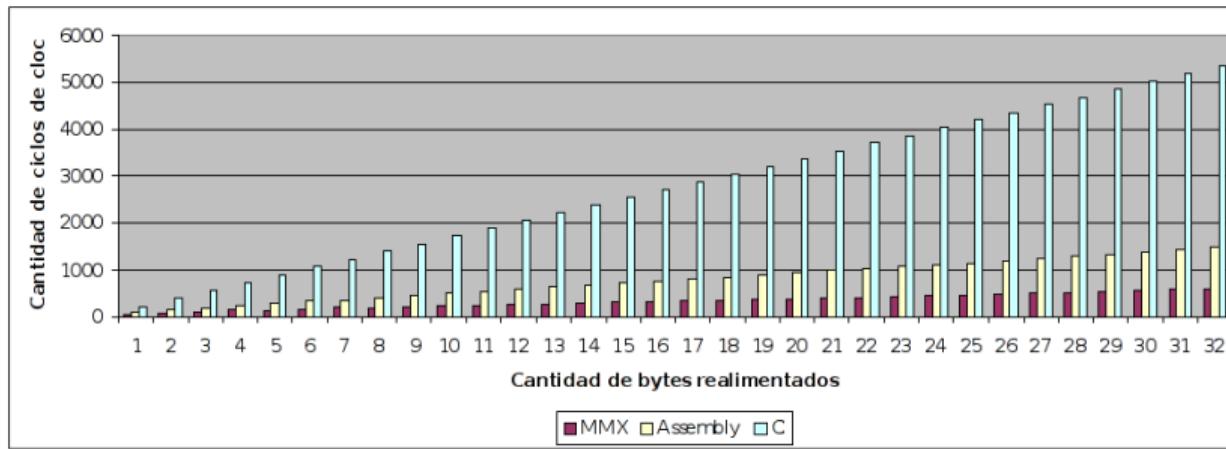
# Resultados

Cantidad de bytes Realimentados	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	256	
Cantidad de clocks por Algoritmo *	MMX	54	69	101	146	130	144	192	178	198	219	235	246	266	282	302	314
	Assembly	91	144	187	231	276	320	364	407	451	496	540	584	627	671	716	760
	C	211	395	559	727	893	1063	1230	1391	1557	1717	1884	2055	2217	2381	2536	2716
		59,3%	47,9%	54,0%	63,2%	47,1%	45,0%	52,7%	43,7%	43,9%	44,2%	43,5%	42,1%	42,4%	42,0%	42,2%	41,3%
		25,6%	17,5%	18,1%	20,1%	14,6%	13,5%	15,6%	12,8%	12,7%	12,8%	12,5%	12,0%	12,0%	11,8%	11,9%	11,6%

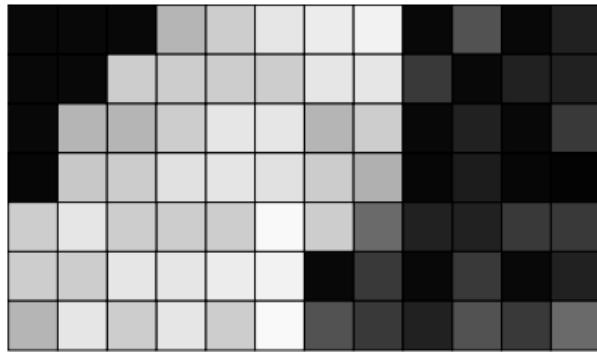
  

Cantidad de bytes Realimentados	272	288	304	320	336	352	368	384	400	416	432	448	464	480	496	512	
Cantidad de clocks por Algoritmo *	MMX	334	349	372	382	404	416	439	450	470	486	510	519	542	554	578	586
	Assembly	804	847	891	936	980	1024	1067	1111	1156	1200	1244	1287	1331	1376	1420	1464
	C	2878	3035	3213	3358	3541	3712	3856	4034	4188	4336	4536	4671	4849	5035	5192	5350
		41,5%	41,2%	41,8%	40,8%	41,2%	40,6%	41,1%	40,5%	40,7%	40,5%	41,0%	40,3%	40,7%	40,3%	40,7%	40,0%
		11,6%	11,5%	11,6%	11,4%	11,4%	11,2%	11,4%	11,2%	11,2%	11,2%	11,2%	11,1%	11,2%	11,0%	11,1%	11,0%

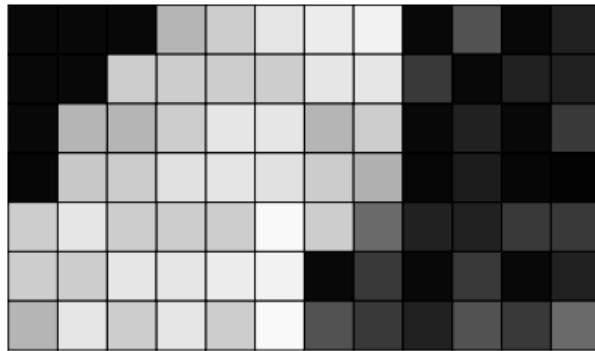
\* Para sistemas no real time es el menor valor de 500 iteraciones del mismo ciclo de acuerdo con lo explicado



# Ejemplo de procesamiento de imágenes



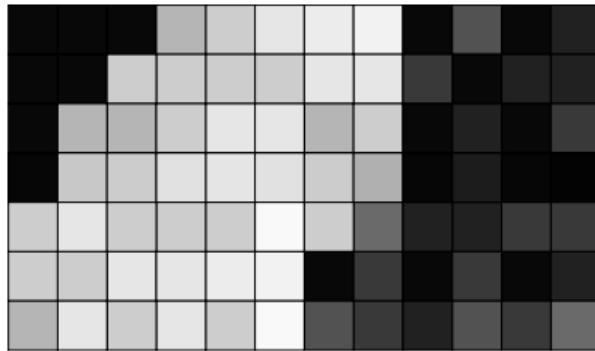
# Ejemplo de procesamiento de imágenes



- Un borde es una transición brusca de iluminación entre dos pixeles vecinos.



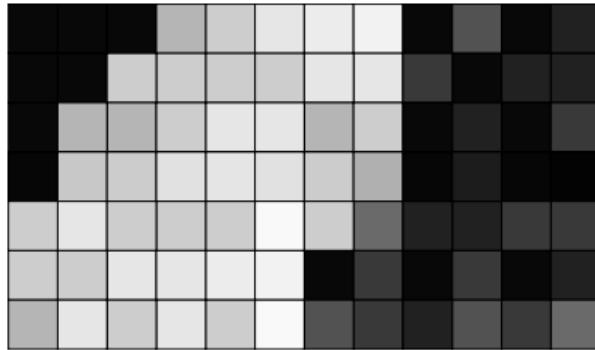
# Ejemplo de procesamiento de imágenes



- Un borde es una transición brusca de iluminación entre dos pixles vecinos.
- En el mapa de bits de la izquierda se muestra como queda una imagen resultante de la detección de bordes



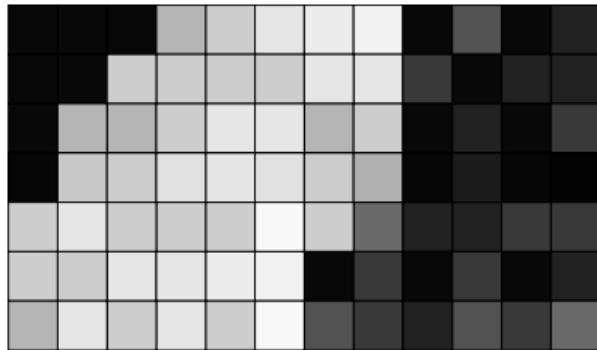
# Ejemplo de procesamiento de imágenes



- Un borde es una transición brusca de iluminación entre dos pixles vecinos.
- En el mapa de bits de la izquierda se muestra como queda una imagen resultante de la detección de bordes
- Un método práctico es por cada pixel tomar los ocho vecinos (N8), y evaluar su diferencia con el mínimo del N8

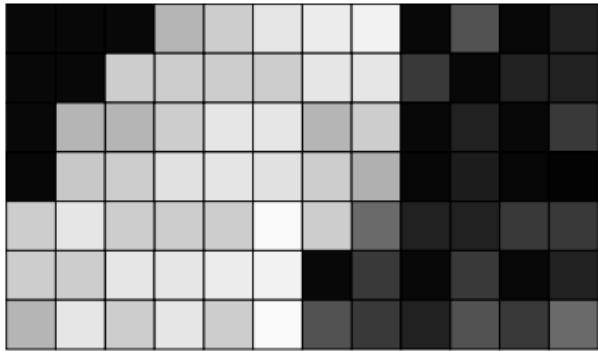


# Ejemplo de procesamiento de imágenes



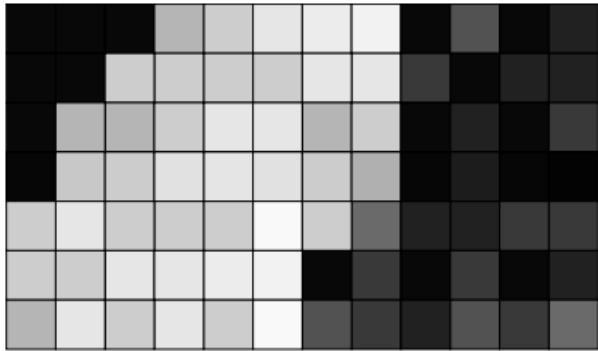
- Un borde es una transición brusca de iluminación entre dos pixles vecinos.
- En el mapa de bits de la izquierda se muestra como queda una imagen resultante de la detección de bordes
- Un método práctico es por cada pixel tomar los ocho vecinos (N8), y evaluar su diferencia con el mínimo del N8
- Si la diferencia es muy grande el valor resultante está próximo al nivel de blanco.

# Ejemplo de procesamiento de imágenes



- Un borde es una transición brusca de iluminación entre dos pixeles vecinos.
- En el mapa de bits de la izquierda se muestra como queda una imagen resultante de la detección de bordes
- Un método práctico es por cada pixel tomar los ocho vecinos ( $N_8$ ), y evaluar su diferencia con el mínimo del  $N_8$
- Si la diferencia es muy grande el valor resultante está próximo al nivel de blanco.
- Si ambos valores tienen una diferencia muy pequeña, esta estará próxima al nivel de negro.

# Ejemplo de procesamiento de imágenes



- Un borde es una transición brusca de iluminación entre dos pixeles vecinos.
- En el mapa de bits de la izquierda se muestra como queda una imagen resultante de la detección de bordes
- Un método práctico es por cada pixel tomar los ocho vecinos ( $N_8$ ), y evaluar su diferencia con el mínimo del  $N_8$
- Si la diferencia es muy grande el valor resultante está próximo al nivel de blanco.
- Si ambos valores tienen una diferencia muy pequeña, esta estará próxima al nivel de negro.
- Reemplazando cada pixel con la diferencia entre su valor y el  $\text{MIN}(N_8)$ , se puede obtener una buena aproximación de los bordes de una imagen

# Ejemplo

```
1 lazo1: movdqu xmm0,[esi+(2*ebx)+2] ; Método cálculo por N8
2     movdqu xmm1,[esi+(2*ebx)+1]
3     movdqu xmm2,[esi+(2*ebx)]
4     movdqu xmm3,[esi+ebx+2]
5     movdqu xmm4,[esi+ebx]
6     movdqu xmm5,[esi+2]
7     movdqu xmm6,[esi+1]
8     movdqu xmm7,[esi+0]
9     pminub xmm0,xmm1           ; Mínimo de los N8 empaquetados en cada registro
10    pminub xmm0,xmm2
11    pminub xmm0,xmm3
12    pminub xmm0,xmm4
13    pminub xmm0,xmm5
14    pminub xmm0,xmm6
15    pminub xmm0,xmm7           ; Mínimo de los N8 en xmm0.
16    movdqu xmm1,[esi+ebx+1]    ; Leo Pixel central
17    pminub xmm0,xmm1           ; Lo computo
18    psbusw xmm1,xmm0          ; Restamos para hallar los bordes
19    movdqu [edi],xmm1
20    add esi,16                 ; Siguiente tira de 16 N8's
21    add edi,16
22    loop lazo1
```

## Resultados con Imágenes: Detección de Bordes

Bordes		
C	ASM	SSE
14269038	9461102	182518
14269584	9461564	182518
14271390	9461494	182518
14271138	9461326	182532
14270690	9461746	182490
14269864	9461438	182504
14270284	9461445	182513,333
1,51	1,00	0,02

# Medición de performance

# Medición de performance

- El recurso que se utiliza es el Registro Model Specific **TSC** (*Time Stamp Counter* introducido a partir del procesador *Pentium*).

# Medición de performance

- El recurso que se utiliza es el Registro Model Specific **TSC** (*Time Stamp Counter* introducido a partir del procesador *Pentium*).
- Ancho: 64 bits.

# Medición de performance

- El recurso que se utiliza es el Registro Model Specific **TSC** (*Time Stamp Counter* introducido a partir del procesador *Pentium*).
- Ancho: 64 bits.
- Se incrementa con cada ciclo de clock del procesador. No hay medida mas “fina” del tiempo de procesamiento que este registro.

# Medición de performance

- El recurso que se utiliza es el Registro Model Specific **TSC** (*Time Stamp Counter* introducido a partir del procesador *Pentium*).
- Ancho: 64 bits.
- Se incrementa con cada ciclo de clock del procesador. No hay medida mas “fina” del tiempo de procesamiento que este registro.
- Para leerlo utilizamos la instrucción rdtsc: `[edx:eax] ← [tsc]`

1 Fundamentos

2 Procesamiento de Señales digitales

3 Modelo de ejecución SIMD

4 Implementaciones SIMD en x86

## 5 Instrucciones

- Transferencias (las mas comunes)
- Aritmética en algoritmos DSP
- **Instrucciones de punto flotante**
- Instrucciones para manejo de enteros para SSEn
- Instrucciones para manejo de cacheabilidad

# Formatos

1 Nro. en Punto Flotante simple precisión escalar

4 Nros. en Punto Flotante simple precisión empaquetados

1 Nro. en Punto Flotante doble precisión escalar

2 Nros. en Punto Flotante doble precisión empaquetados

- Las mismas operaciones en cada formato se manejan con diferentes instrucciones en cada caso.

# 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

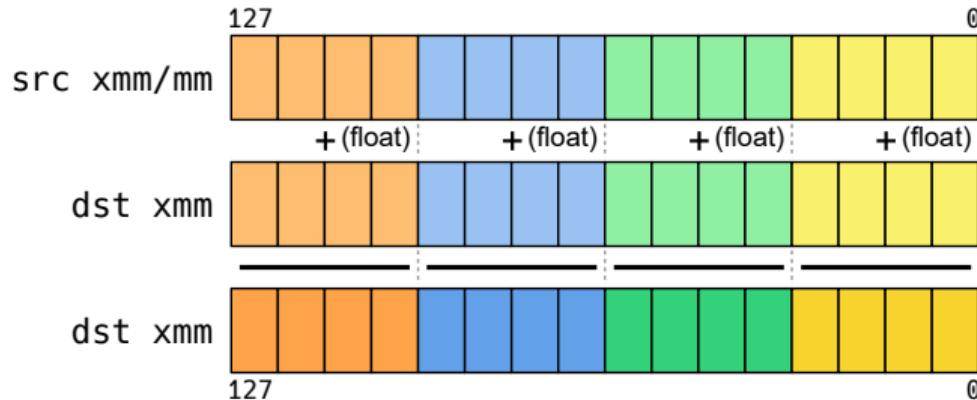
Ejemplos:

# 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] ✓

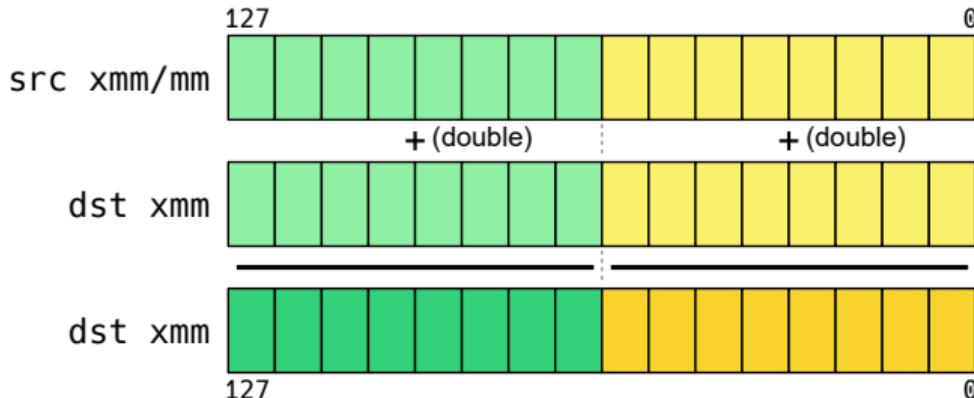


# Operaciones Aritméticas

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] ✓

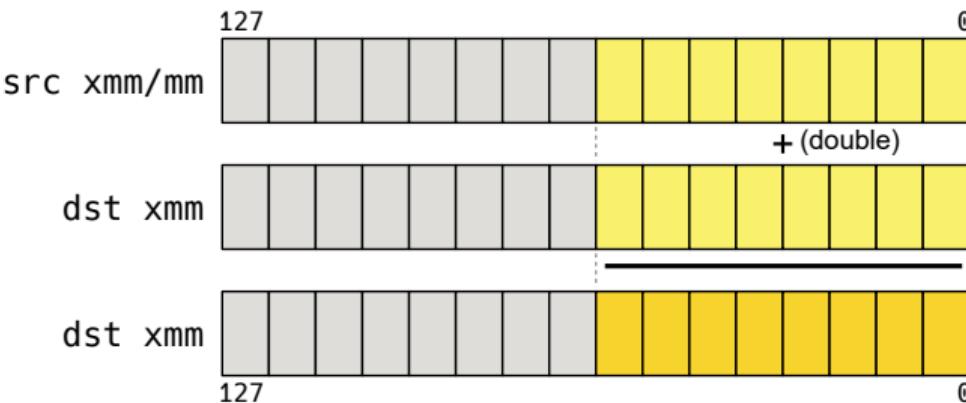


# Operaciones Aritméticas

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

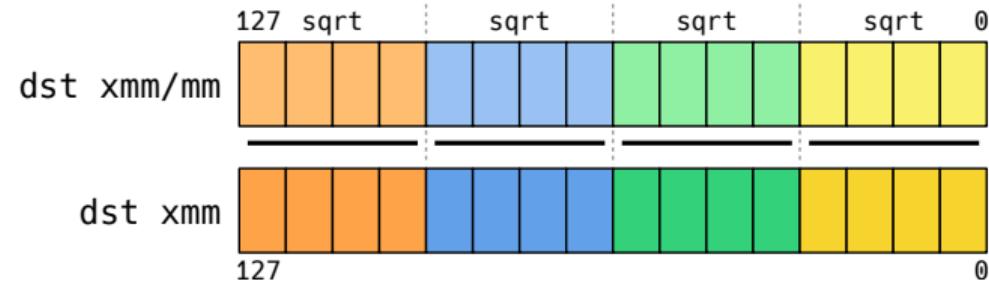
---

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 |
    psrlqd 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 |
    psrlqd 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 |
    psrlqd 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 |
psrlfdq 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	
psrlfdq xmm1, 4 ; xmm1 =   0   max   0   0	
por xmm1, xmm0 ; xmm1 =   max   max   0   0	
movdqu xmm0, xmm1 ; xmm0 =   max   max   0   0	
psrlfdq 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 |
psrlfdq xmm2, 4   ; xmm2 = | 0 | min | 0 | 0 |
por    xmm2, xmm0 ; xmm2 = | min | min | 0 | 0 |
movdqu xmm0, xmm2 ; xmm0 = | min | min | 0 | 0 |
psrlfdq 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

```

## Instrucciones Lógicas

Todas consideran a los operandos como un vector de bits, de modo que no tienen diferencia con las instrucciones lógicas convencionales. Se las denomina operaciones “bitwise”.

- **ANDPS** Realiza una AND lógica entre dos valores de punto flotante single-precision empaquetados.
- **ANDNPS** Realiza una NAND lógica entre dos valores de punto flotante single-precision empaquetados.
- **ORPS** Realiza una OR lógica entre dos valores de punto flotante single-precision empaquetados.
- **XORPS** Realiza una XOR lógica entre dos valores de punto flotante single-precision empaquetados.

# Instrucciones Lógicas

- **ANDPD** Realiza una AND lógica entre dos valores de punto flotante doble precisión empaquetados.
- **ANDNPD** Realiza una NAND lógica entre dos valores de punto flotante doble precisión empaquetados.
- **ORPD** Realiza una OR lógica entre dos valores de punto flotante doble precisión empaquetados.
- **XORPD** Realiza una XOR lógica entre dos valores de punto flotante doble precisión empaquetados.

# Instrucciones de Comparación

Realizan comparaciones entre operandos y afectan al operando destino o al registro EFLAGS.

- **CMPPS** Compara valores empaquetados de punto flotante de simple precisión.
- **CMPPD** Compara valores empaquetados de punto flotante doble precisión.
- **CMPSS** Compara escalares de punto flotante simple precisión.
- **CMPSD** Compara escalares de punto flotante doble precisión.
- **COMISS** Compara los valores escalares de punto flotante simple precisión y modifica flags en el registro EFLAGS. Genera excepción #IU (Invalid Operation), tanto si uno de los operandos es SNaN como QNaN.
- **UCOMISS** Es igual a **COMISS** pero solo genera excepción #IU si uno de los operandos es SNaN.
- **COMISD** y **UCOMISD** son las versiones para escalares doble precisión de **COMISS** y **UCOMISS**.

# Instrucciones de Comparación

La comparación en detalle lleva tres operandos: Dos registros **XMM** o un registro **XMM** y una dirección de memoria, y un tercer operando que es un byte, cuyo valor define hasta 8 alternativas. Cada dword del operando destino será **0xFFFFFFFF** si el resultado de la comparación es **TRUE** y **0x00000000** si es **FALSE**.

- **CMPEQPS** **xmm1, xmm2;** compara si es igual a...;
- **CMPLTPS** **xmm1, xmm2;** compara si es menor que...;
- **CMPLEPS** **xmm1, xmm2;** compara si es menor o igual a...;
- **CMPUNORDPS** **xmm1, xmm2;** compara si está desordenado (cada operando == NaN);
- **CMPNEQPS** **xmm1, xmm2;** compara si no es igual a...;
- **CMPNLTSPS** **xmm1, xmm2;** compara si no es menor que...;
- **CMPNLEPS** **xmm1, xmm2;** compara si no es menor o igual a...;
- **CMPORDPS** **xmm1, xmm2;** compara si está ordenado (cada operando != NaN);

# Instrucciones de Comparación

Para escalares Simple precisión.

- **CMPEQSS**      `xmm1, xmm2`; compara si es igual a...;
- **CMPLTSS**      `xmm1, xmm2`; compara si es menor que...;
- **CMPLESS**      `xmm1, xmm2`; compara si es menor o igual a...;
- **CMPUNORDSS** `xmm1, xmm2`; compara si está desordenado (dword baja == NaN);
- **CMPNEQSS**      `xmm1, xmm2`; compara si no es igual a....;
- **CMPNLTSS**      `xmm1, xmm2`; compara si no es menor que...;
- **CMPNLESS**      `xmm1, xmm2`; compara si no es menor o igual a...;
- **CMPORDSS**      `xmm1, xmm2`; compara si está ordenado (dword baja != NaN);

# Instrucciones de Comparación

Para Punto flotante empaquetado de doble precisión.

- **CMPEQPD**  $\text{xmm1}, \text{xmm2}$ ; compara si es igual a...;
- **CMPLTPD**  $\text{xmm1}, \text{xmm2}$ ; compara si es menor que...;
- **CMPLEPD**  $\text{xmm1}, \text{xmm2}$ ; compara si es menor o igual a...;
- **CMPUNORDPD**  $\text{xmm1}, \text{xmm2}$ ; compara si está desordenado (cada operando == NaN);
- **CMPNEQPD**  $\text{xmm1}, \text{xmm2}$ ; compara si no es igual a...;
- **CMPNLTPD**  $\text{xmm1}, \text{xmm2}$ ; compara si no es menor que...;
- **CMPNLEPD**  $\text{xmm1}, \text{xmm2}$ ; compara si no es menor o igual a...;
- **CMPORDPD**  $\text{xmm1}, \text{xmm2}$ ; compara si está ordenado (cada operando != NaN);

# Instrucciones de Comparación

Para escalares Doble precisión.

- **CMPEQSD**       $\text{xmm1}, \text{xmm2}$ ; compara si es igual a...
- **CMLTSD**       $\text{xmm1}, \text{xmm2}$ ; compara si es menor que...
- **CMPLESND**       $\text{xmm1}, \text{xmm2}$ ; compara si es menor o igual a...
- **CMPUNORDSD**       $\text{xmm1}, \text{xmm2}$ ; compara si está desordenado (qword baja == NaN)
- **CMPNEQSD**       $\text{xmm1}, \text{xmm2}$ ; compara si no es igual a...
- **CMPNLTSND**       $\text{xmm1}, \text{xmm2}$ ; compara si no es menor que...
- **CMPNLESND**       $\text{xmm1}, \text{xmm2}$ ; compara si no es menor o igual a...
- **CMPORDSD**       $\text{xmm1}, \text{xmm2}$ ; compara si está ordenado (qword baja != NaN)

# Shuffles

Las instrucciones de *Shuffle* permiten **reordenar** datos en registros.

Sus parámetros serán el **registro a reordenar** y una **máscara** que indicará cómo hacerlo.

# Shuffles

Las instrucciones de *Shuffle* permiten **reordenar** datos en registros.

Sus parámetros serán el **registro a reordenar** y una **máscara** que indicará cómo hacerlo.

- PSHUFB - *Shuffle Packed Bytes*
- PSHUFHW - *Shuffles high 16bit values*
- PSHUFLW - *Shuffles low 16bit values*
- PSHUFD - *Shuffle Packed Doublewords*
- SHUFPS - *Shuffle Packed Single FP Values*
- SHUFPD - *Shuffle Packed Double FP Values*

# Shuffles

## PSHUFB – Packed Shuffle Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 38 00 /r <sup>1</sup> PSHUFB <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Shuffle bytes in <i>mm1</i> according to contents of <i>mm2/m64</i> .
66 OF 38 00 /r PSHUFB <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Shuffle bytes in <i>xmm1</i> according to contents of <i>xmm2/m128</i> .
VEX.NDS.128.66.0F38.WIG 00 /r VPSHUFB <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Shuffle bytes in <i>xmm2</i> according to contents of <i>xmm3/m128</i> .
VEX.NDS.256.66.0F38.WIG 00 /r VPSHUFB <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Shuffle bytes in <i>ymm2</i> according to contents of <i>ymm3/m256</i> .

# Shuffles

## PSHUFB – Packed Shuffle Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 38 00 /r <sup>1</sup> PSHUFB <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Shuffle bytes in <i>mm1</i> according to contents of <i>mm2/m64</i> .
66 OF 38 00 /r PSHUFB <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Shuffle bytes in <i>xmm1</i> according to contents of <i>xmm2/m128</i> .
VEX.NDS.128.66.0F38.WIG 00 /r VPSHUFB <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Shuffle bytes in <i>xmm2</i> according to contents of <i>xmm3/m128</i> .
VEX.NDS.256.66.0F38.WIG 00 /r VPSHUFB <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Shuffle bytes in <i>ymm2</i> according to contents of <i>ymm3/m256</i> .

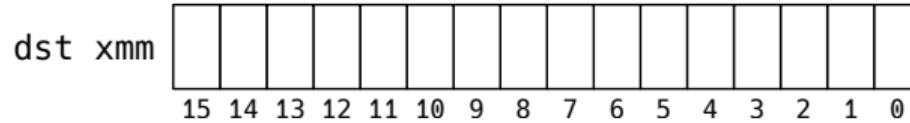
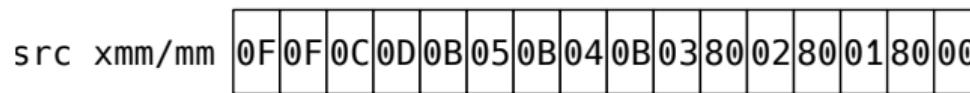
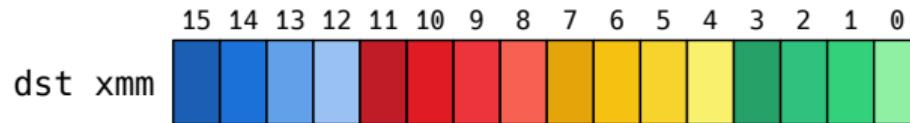
### PSHUFB (with 128 bit operands)

```

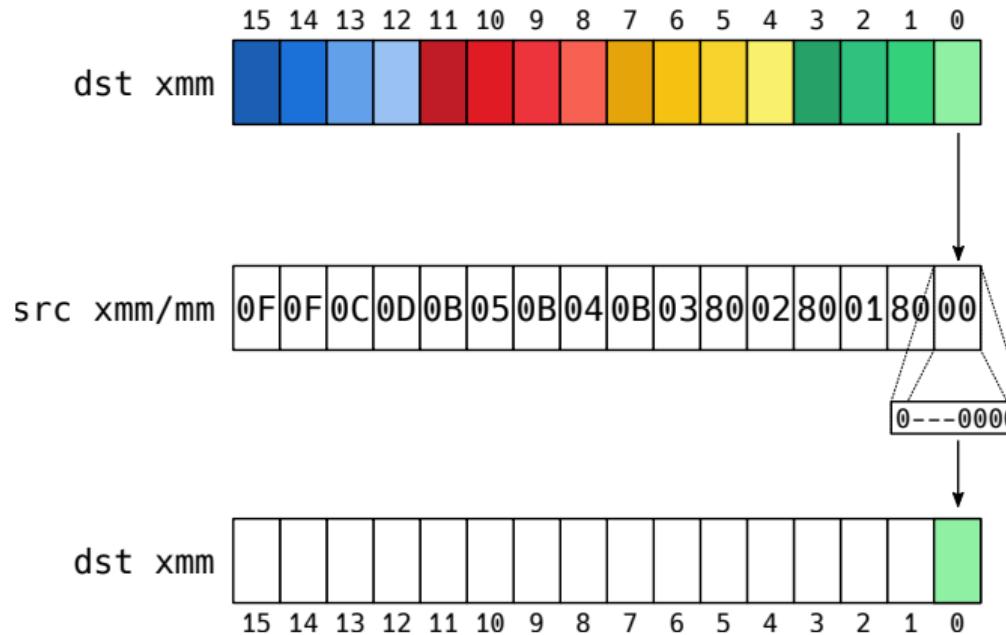
for i = 0 to 15 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7..(i*8)+0] ← 0;
    else
        index[3..0] ← SRC[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← DEST[(index*8+7)..(index*8+0)];
    endif
}
DEST[VLMAX-1:128] ← 0

```

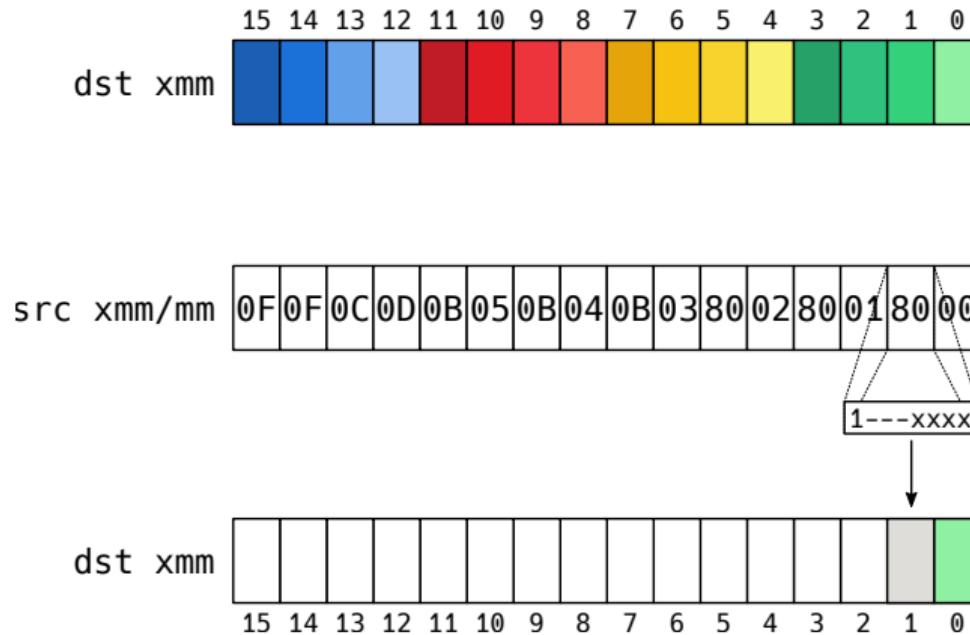
# Ejemplo - PSHUFB dst, src



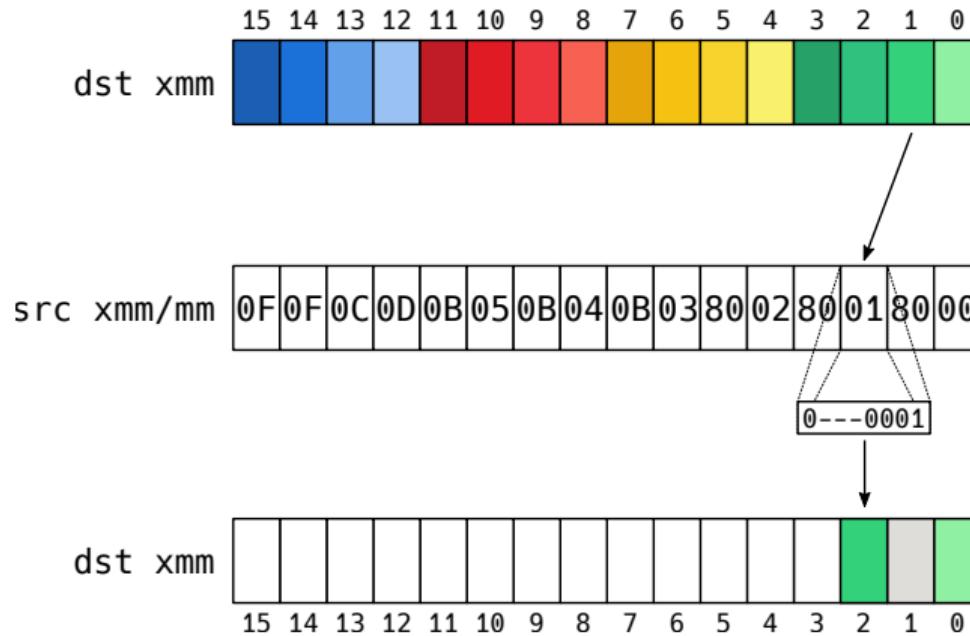
## Ejemplo - PSHUFB dst, src



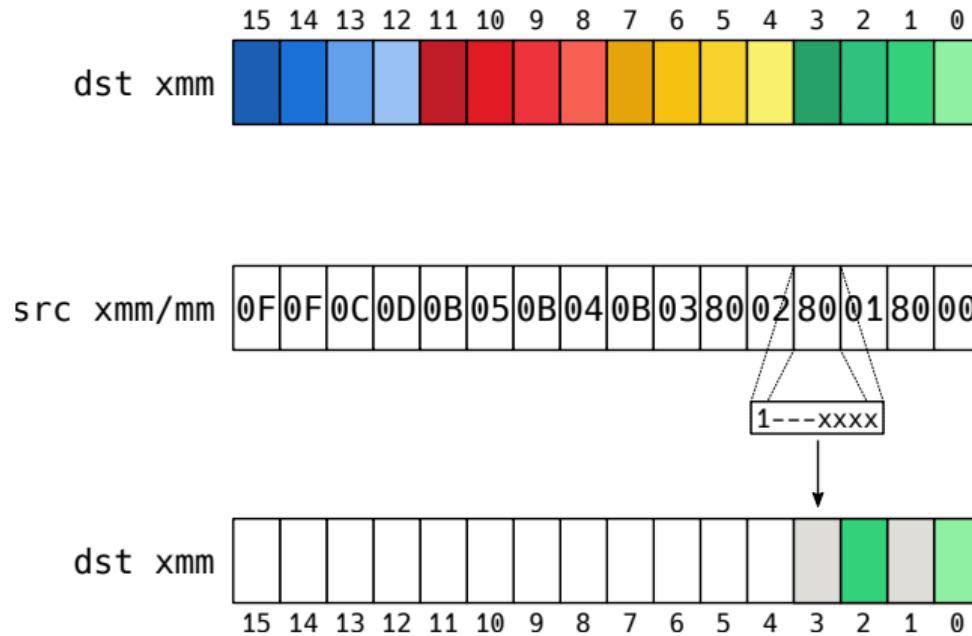
## Ejemplo - PSHUFB dst, src



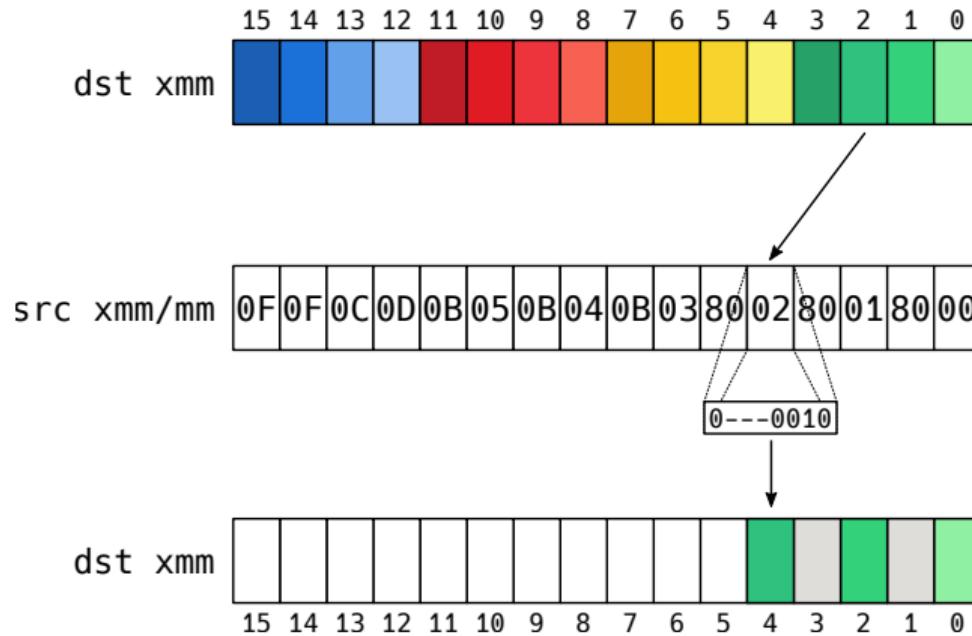
## Ejemplo - PSHUFB dst, src



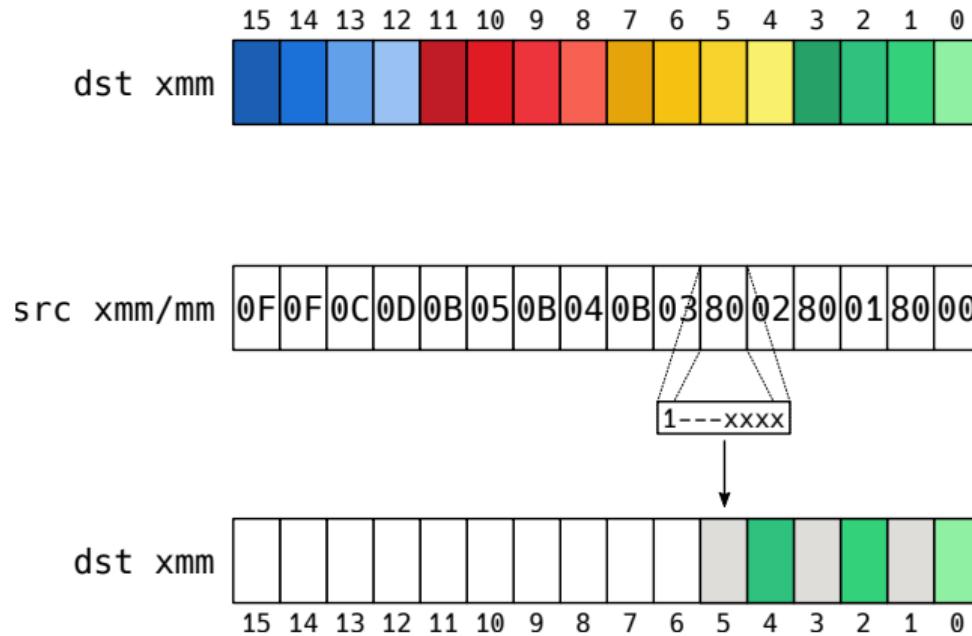
## Ejemplo - PSHUFB dst, src



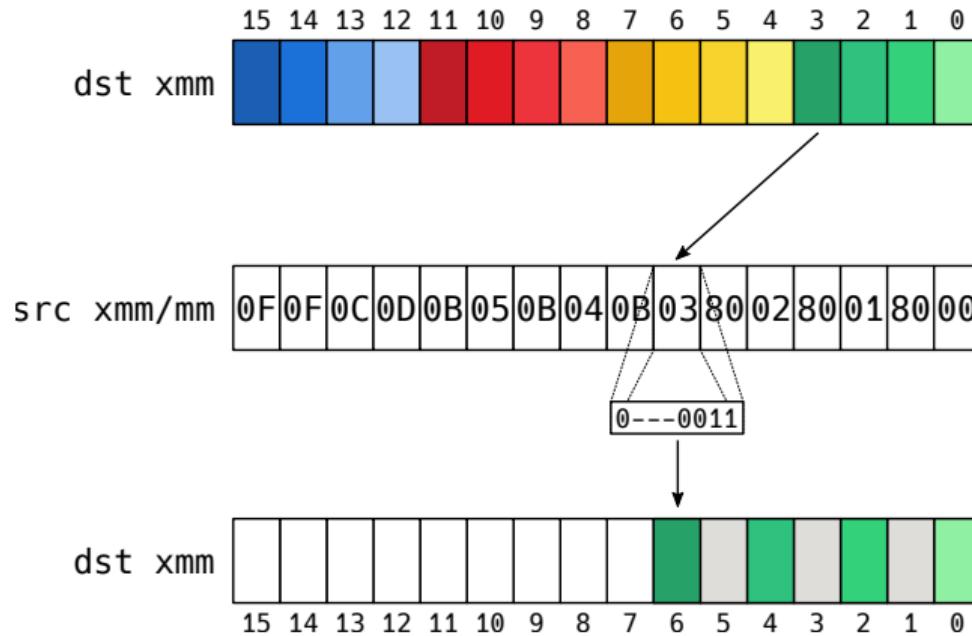
## Ejemplo - PSHUFB dst, src



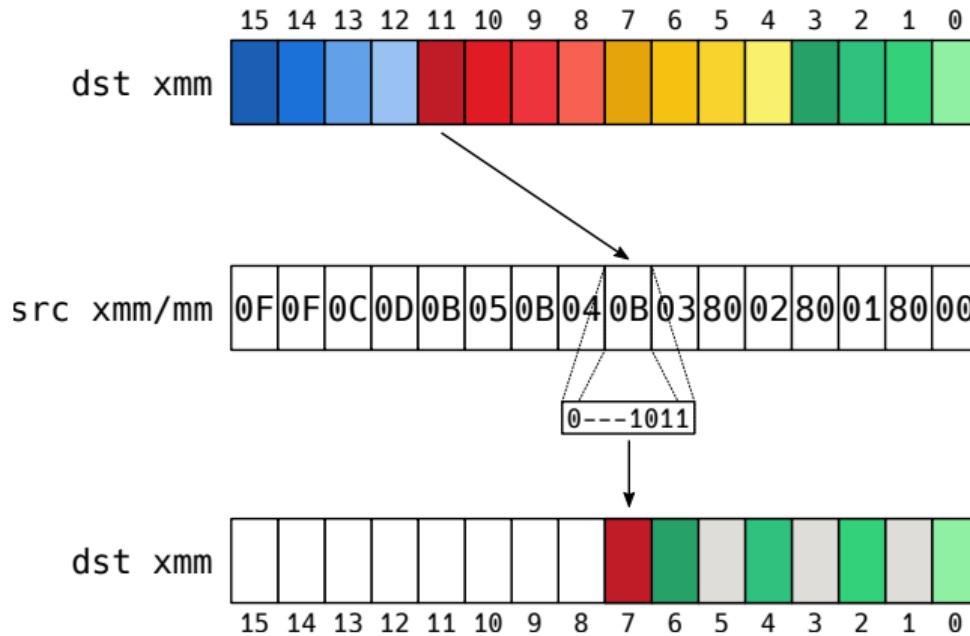
## Ejemplo - PSHUFB dst, src



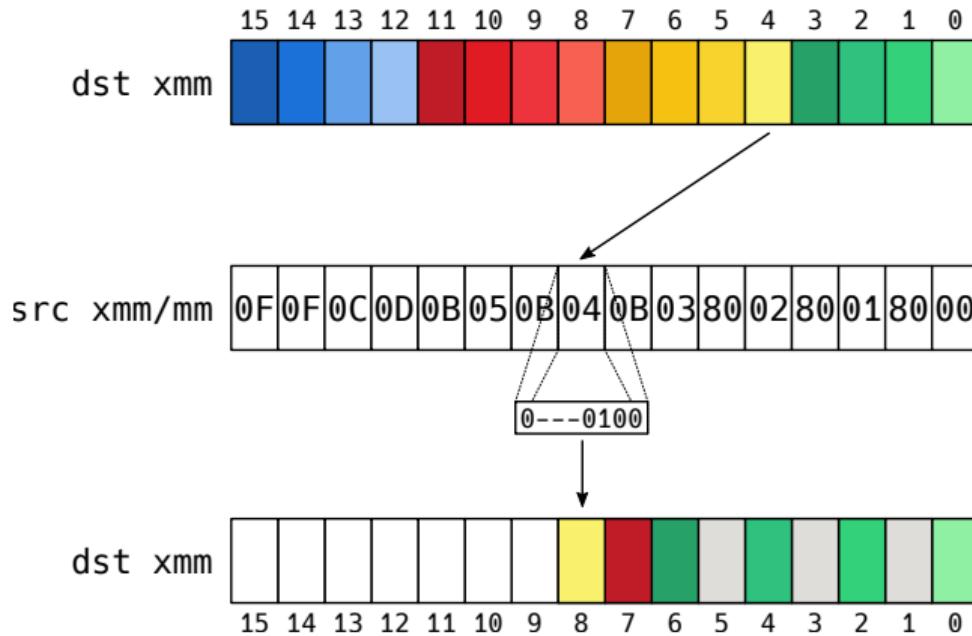
## Ejemplo - PSHUFB dst, src



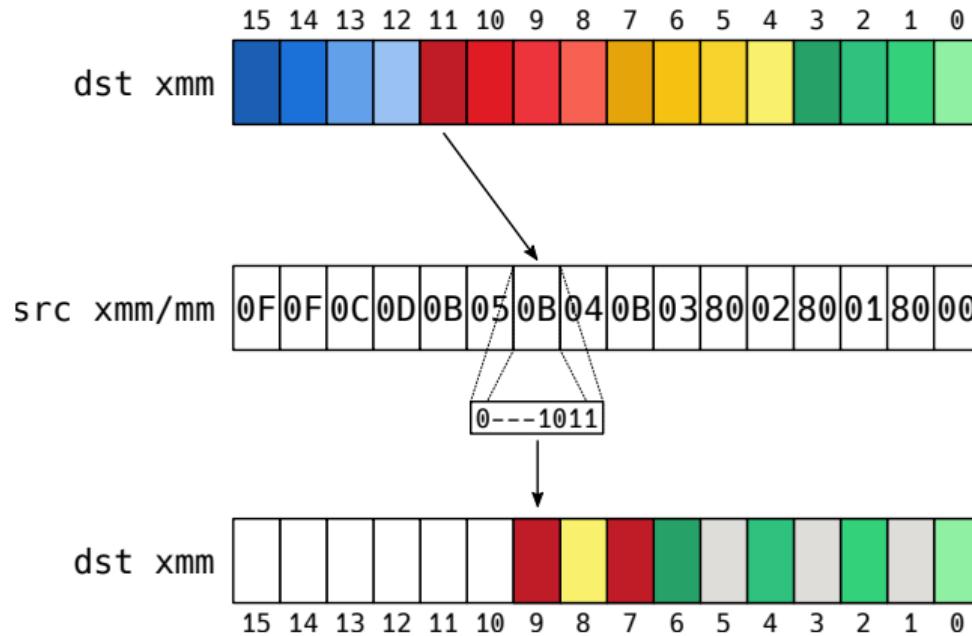
## Ejemplo - PSHUFB dst, src



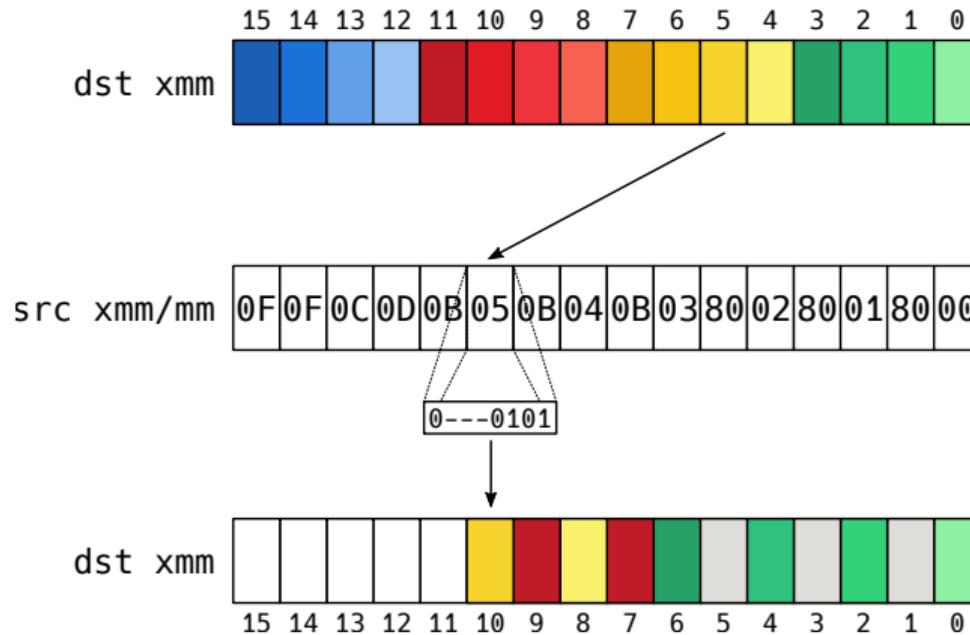
## Ejemplo - PSHUFB dst, src



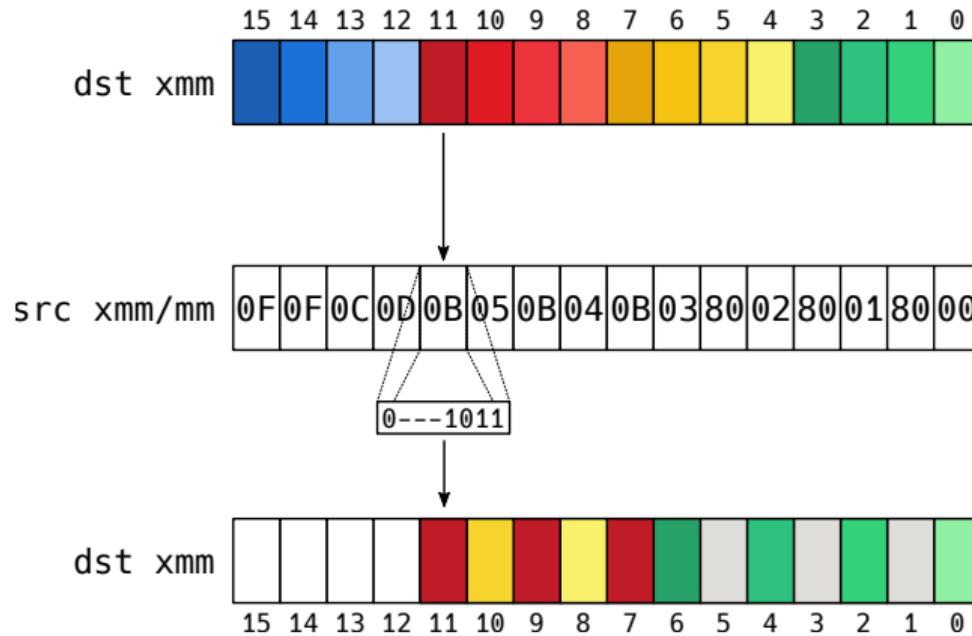
## Ejemplo - PSHUFB dst, src



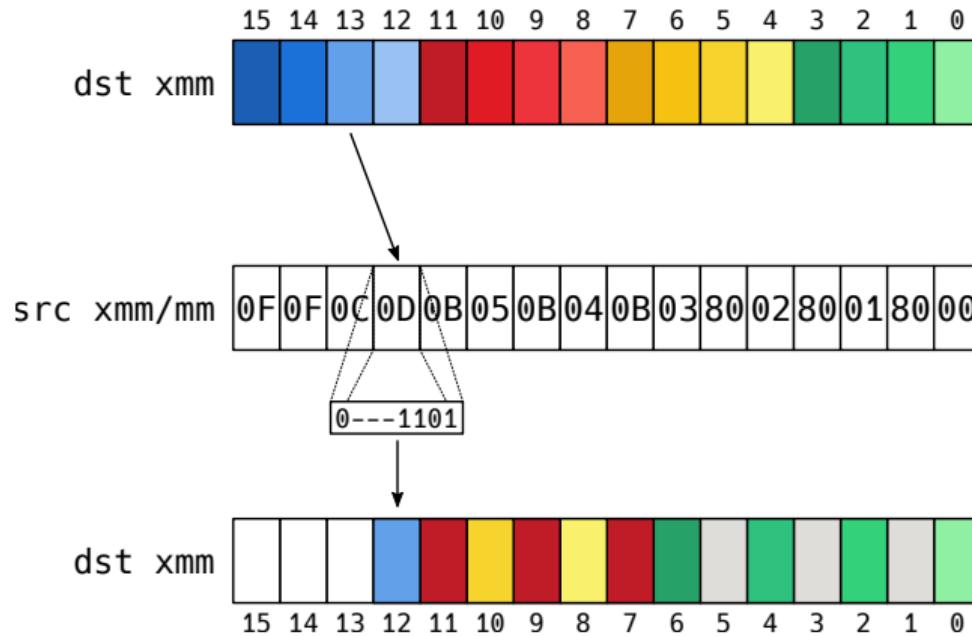
## Ejemplo - PSHUFB dst, src



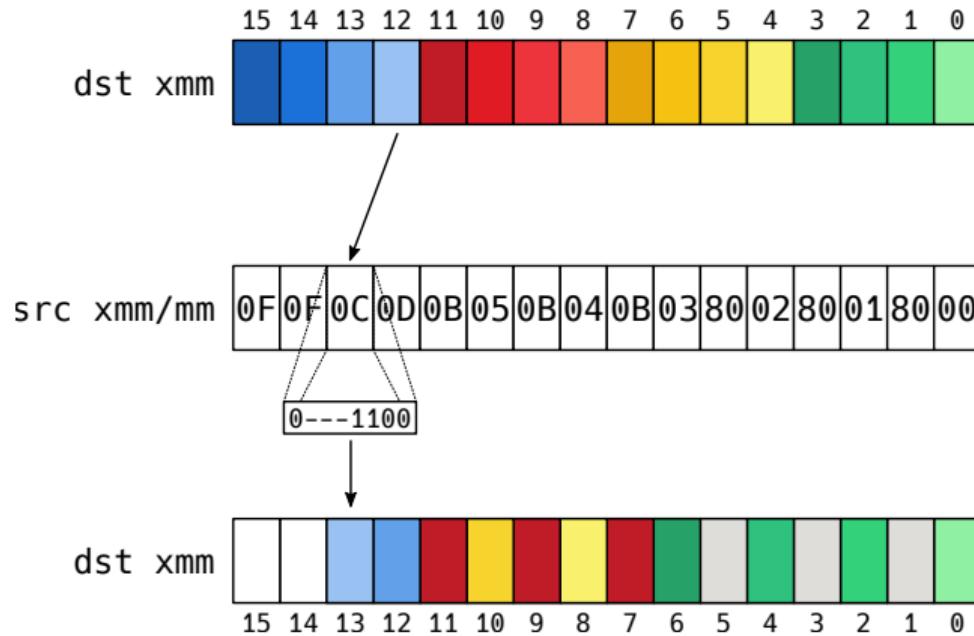
## Ejemplo - PSHUFB dst, src



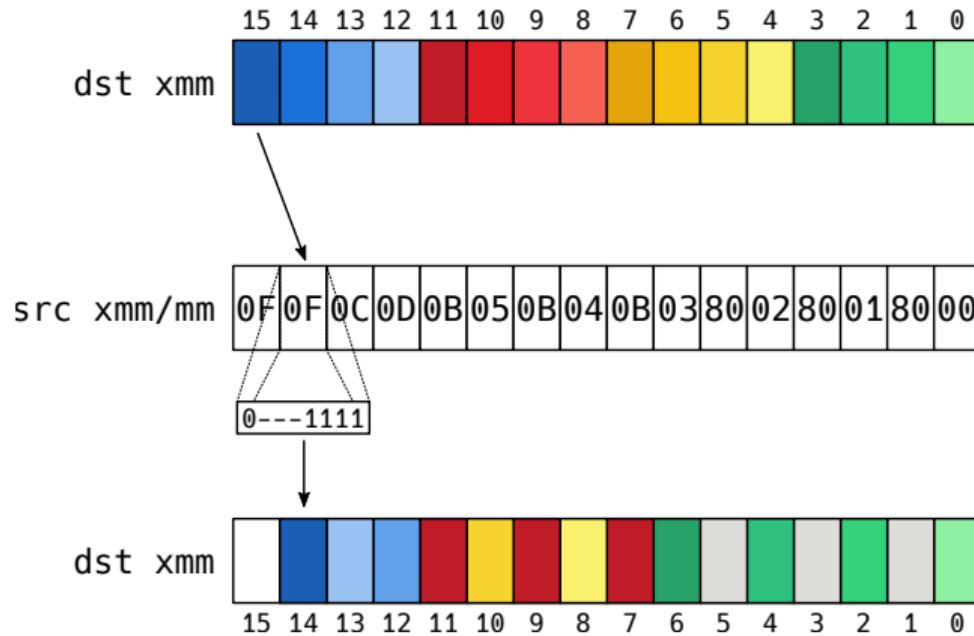
## Ejemplo - PSHUFB dst, src



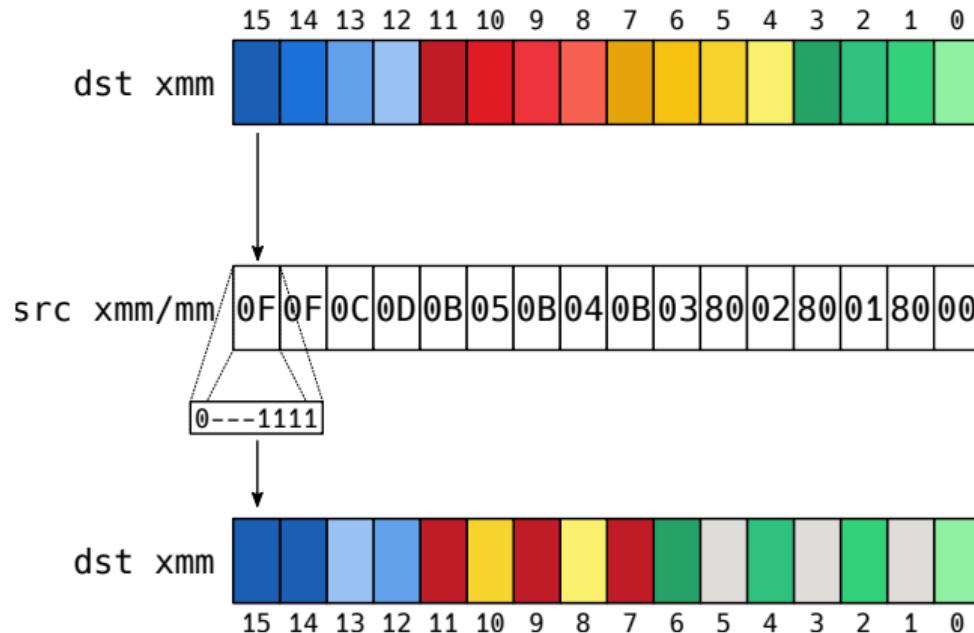
## Ejemplo - PSHUFB dst, src



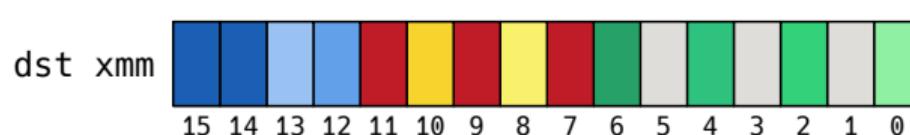
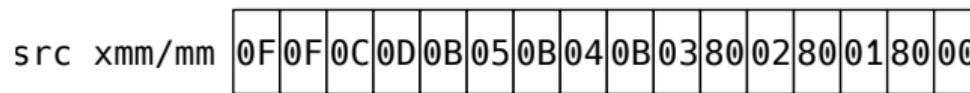
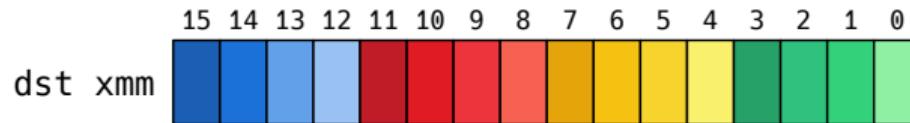
## Ejemplo - PSHUFB dst, src



## Ejemplo - PSHUFB dst, src



## Ejemplo - PSHUFB dst, src



# Shuffles

## PSHUFLW—Shuffle Packed Low Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 OF 70 /r ib PSHUFLW <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE2	Shuffle the low words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.F2.0F.WIG 70 /r ib VPSHUFLW <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Shuffle the low words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.F2.0F.WIG 70 /r ib VPSHUFLW <i>ymm1, ymm2/m256, imm8</i>	RMI	V/V	AVX2	Shuffle the low words in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .

# Shuffles

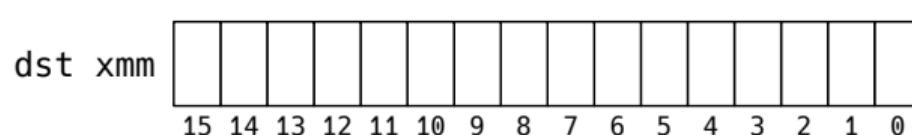
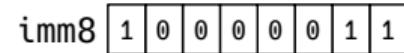
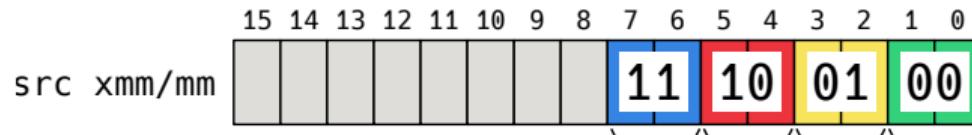
## PSHUFLW—Shuffle Packed Low Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 OF 70 /r ib PSHUFLW <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE2	Shuffle the low words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.F2.0F.WIG 70 /r ib VPSHUFLW <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Shuffle the low words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.F2.0F.WIG 70 /r ib VPSHUFLW <i>ymm1, ymm2/m256, imm8</i>	RMI	V/V	AVX2	Shuffle the low words in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .

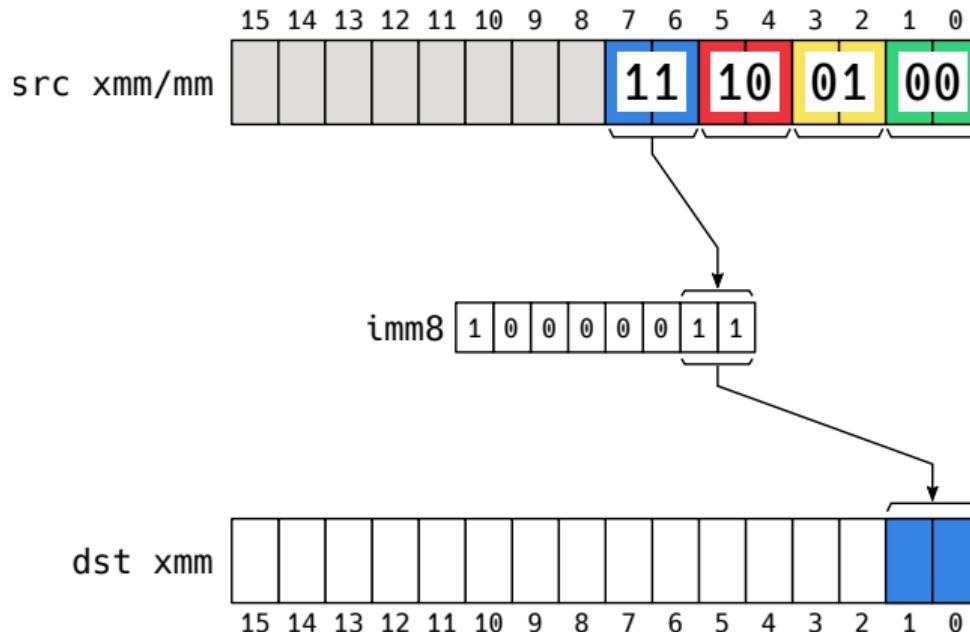
### PSHUFLW (128-bit Legacy SSE version)

DEST[15:0]  $\leftarrow$  (SRC  $\gg$  (imm[1:0] \* 16))[15:0]  
 DEST[31:16]  $\leftarrow$  (SRC  $\gg$  (imm[3:2] \* 16))[15:0]  
 DEST[47:32]  $\leftarrow$  (SRC  $\gg$  (imm[5:4] \* 16))[15:0]  
 DEST[63:48]  $\leftarrow$  (SRC  $\gg$  (imm[7:6] \* 16))[15:0]  
 DEST[127:64]  $\leftarrow$  SRC[127:64]  
 DEST[VLMAX-1:128] (Unmodified)

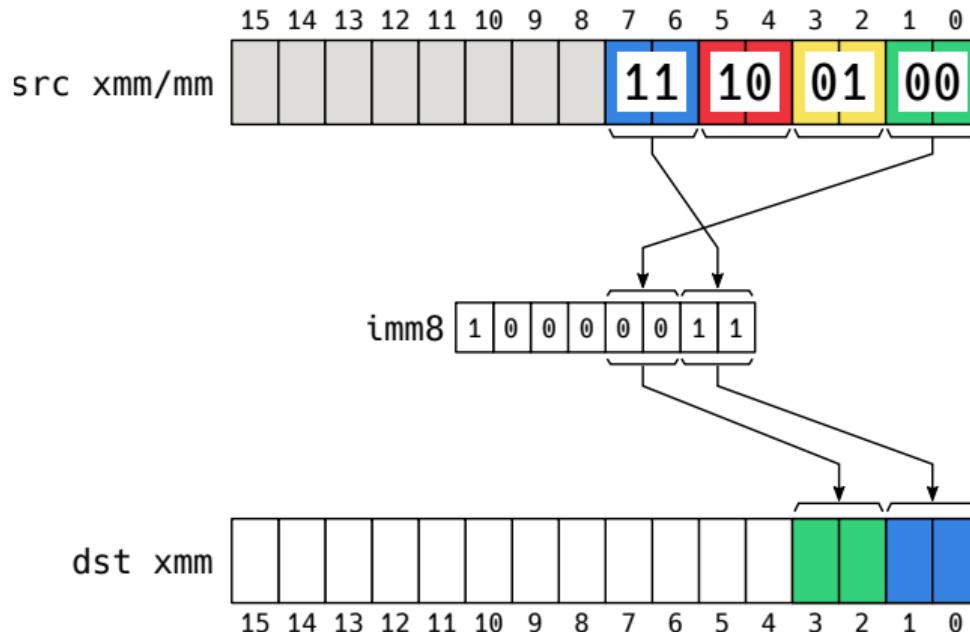
## Ejemplo - PSHUFLW dst, src , imm8



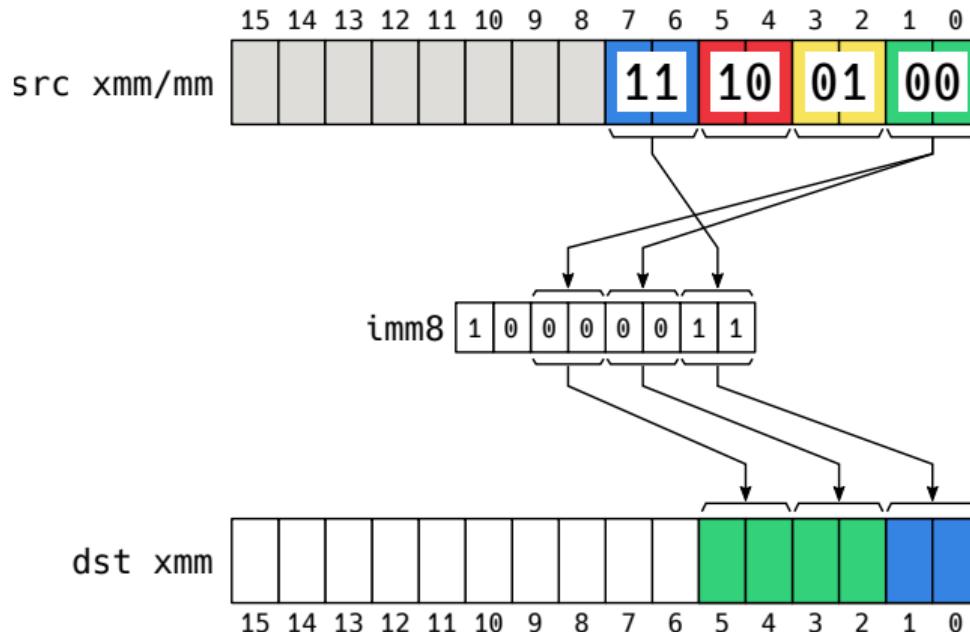
## Ejemplo - PSHUFLW dst, src , imm8



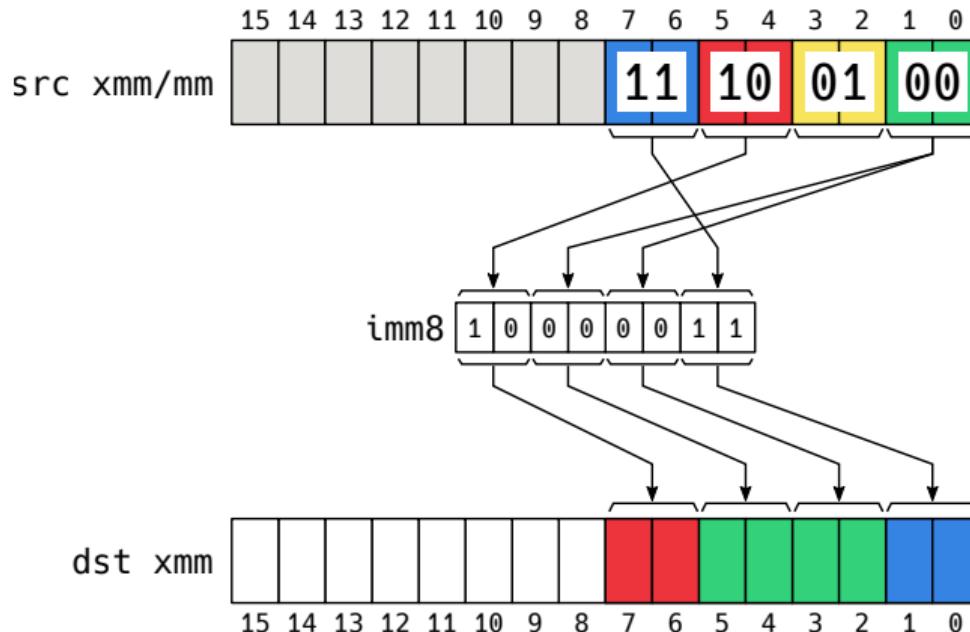
## Ejemplo - PSHUFLW dst, src , imm8



## Ejemplo - PSHUFLW dst, src , imm8



## Ejemplo - PSHUFLW dst, src , imm8



# Shuffles

## PSHUFHW—Shuffle Packed High Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 OF 70 /r ib PSHUFHW <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE2	Shuffle the high words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.F3.0F.WIG 70 /r ib VPSHUFHW <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Shuffle the high words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.F3.0F.WIG 70 /r ib VPSHUFHW <i>ymm1, ymm2/m256, imm8</i>	RMI	V/V	AVX2	Shuffle the high words in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .

# Shuffles

## PSHUFHW—Shuffle Packed High Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 OF 70 /r ib PSHUFHW <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE2	Shuffle the high words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.F3.0F.WIG 70 /r ib VPSHUFHW <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Shuffle the high words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.F3.0F.WIG 70 /r ib VPSHUFHW <i>ymm1, ymm2/m256, imm8</i>	RMI	V/V	AVX2	Shuffle the high words in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .

### PSHUFHW (128-bit Legacy SSE version)

$$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$$

$$\text{DEST}[79:64] \leftarrow (\text{SRC} \gg (\text{imm}[1:0] * 16))[79:64]$$

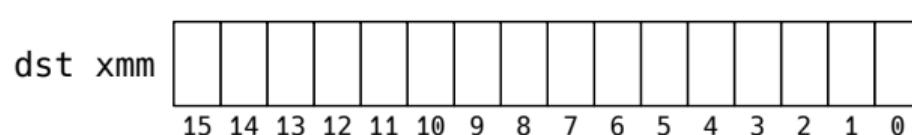
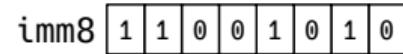
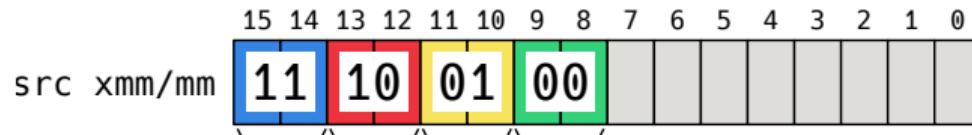
$$\text{DEST}[95:80] \leftarrow (\text{SRC} \gg (\text{imm}[3:2] * 16))[79:64]$$

$$\text{DEST}[111:96] \leftarrow (\text{SRC} \gg (\text{imm}[5:4] * 16))[79:64]$$

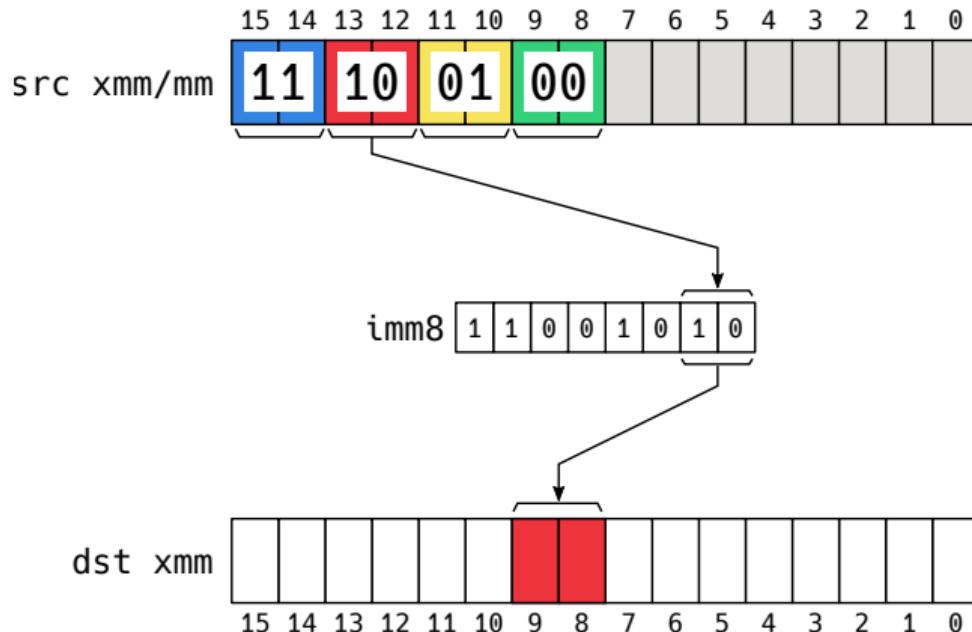
$$\text{DEST}[127:112] \leftarrow (\text{SRC} \gg (\text{imm}[7:6] * 16))[79:64]$$

DEST[VLMAX-1:128] (Unmodified)

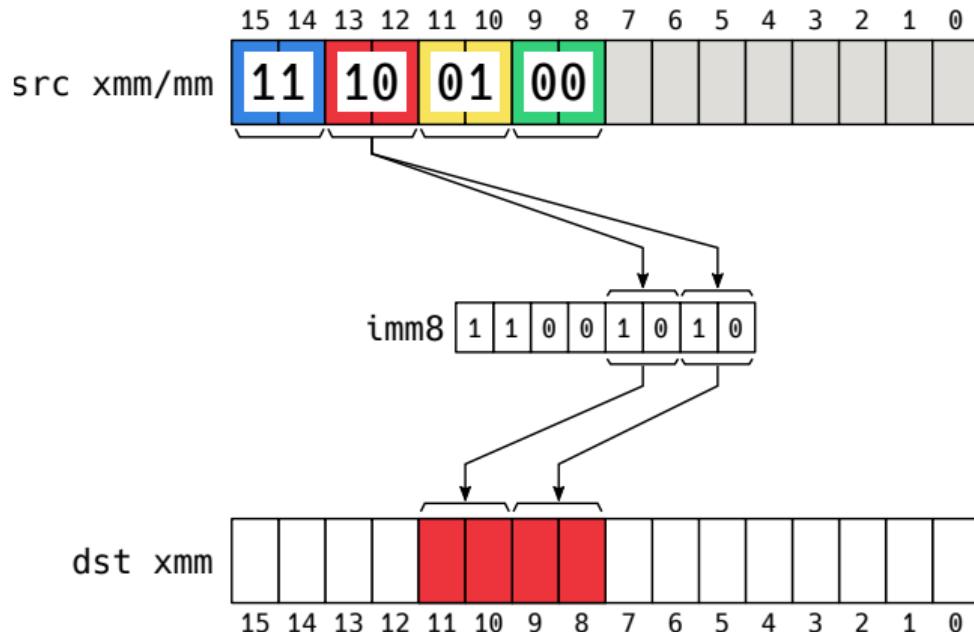
## Ejemplo - PSHUFW dst, src , imm8



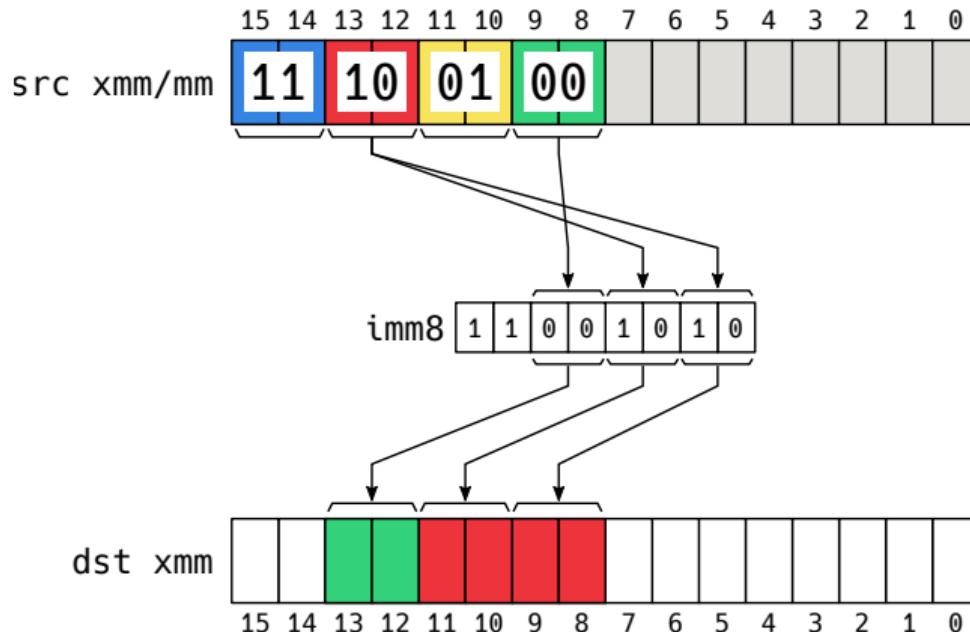
## Ejemplo - PSHUFWH dst, src , imm8



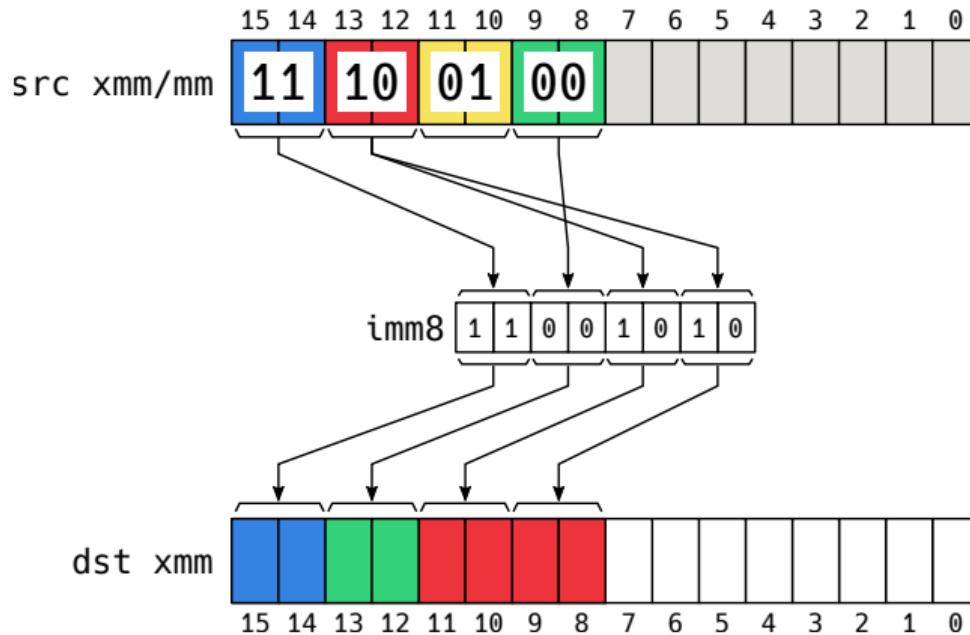
## Ejemplo - PSHUFWH dst, src , imm8



## Ejemplo - PSHUFW dst, src , imm8



## Ejemplo - PSHUFW dst, src , imm8



# Shuffles

## PSHUFD—Shuffle Packed Doublewords

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 OF 70 /rib PSHUFD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE2	Shuffle the doublewords in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.66.0F.WIG 70 /r ib VPSHUFD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Shuffle the doublewords in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.66.0F.WIG 70 /r ib VPSHUFD <i>ymm1, ymm2/m256, imm8</i>	RMI	V/V	AVX2	Shuffle the doublewords in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .

# Shuffles

## PSHUFD—Shuffle Packed Doublewords

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 OF 70 /rib PSHUFD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE2	Shuffle the doublewords in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.66.0F.WIG 70 /r ib VPSHUFD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Shuffle the doublewords in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.66.0F.WIG 70 /r ib VPSHUFD <i>ymm1, ymm2/m256, imm8</i>	RMI	V/V	AVX2	Shuffle the doublewords in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .

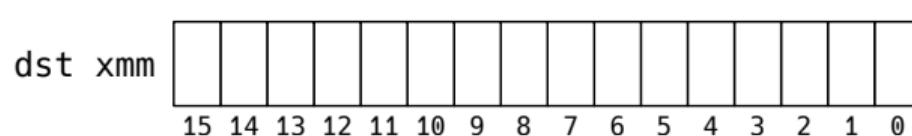
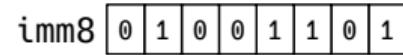
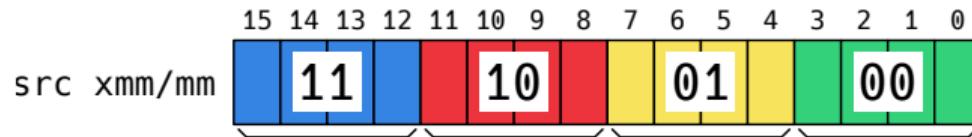
### PSHUFD (128-bit Legacy SSE version)

```

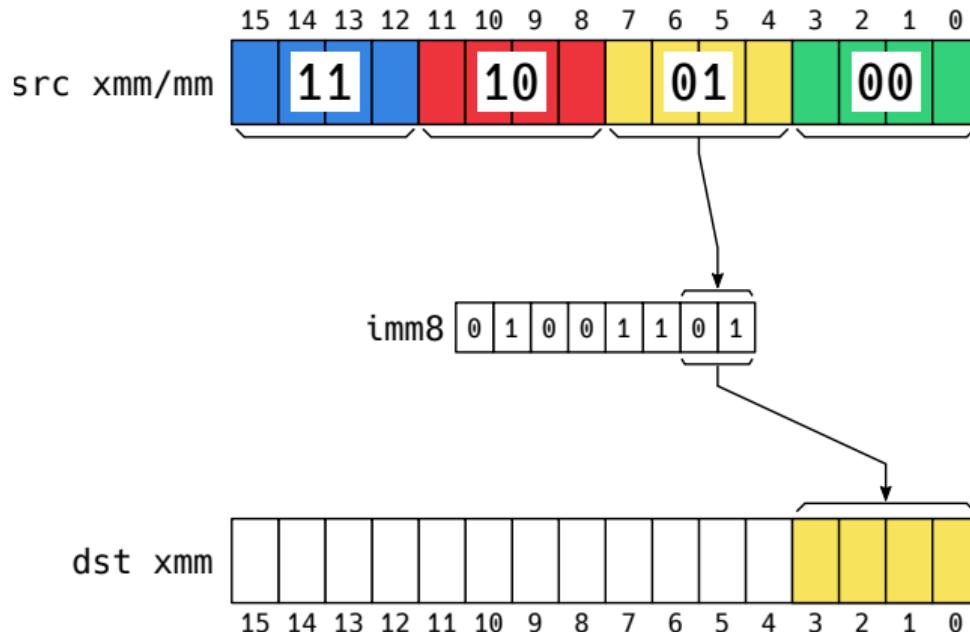
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[VLMAX-1:128] (Unmodified)

```

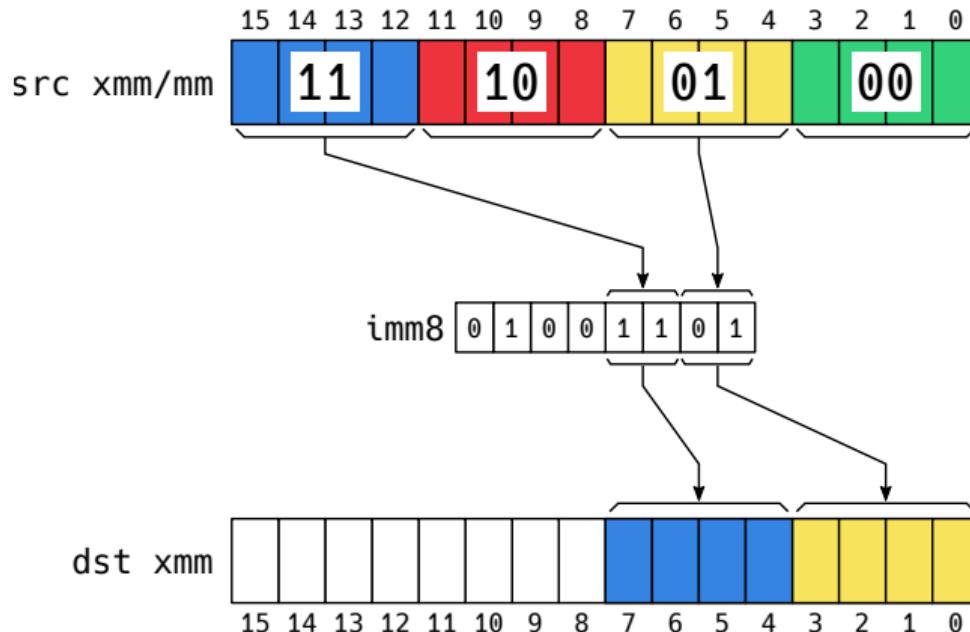
## Ejemplo - PSHUFD dst, src , imm8



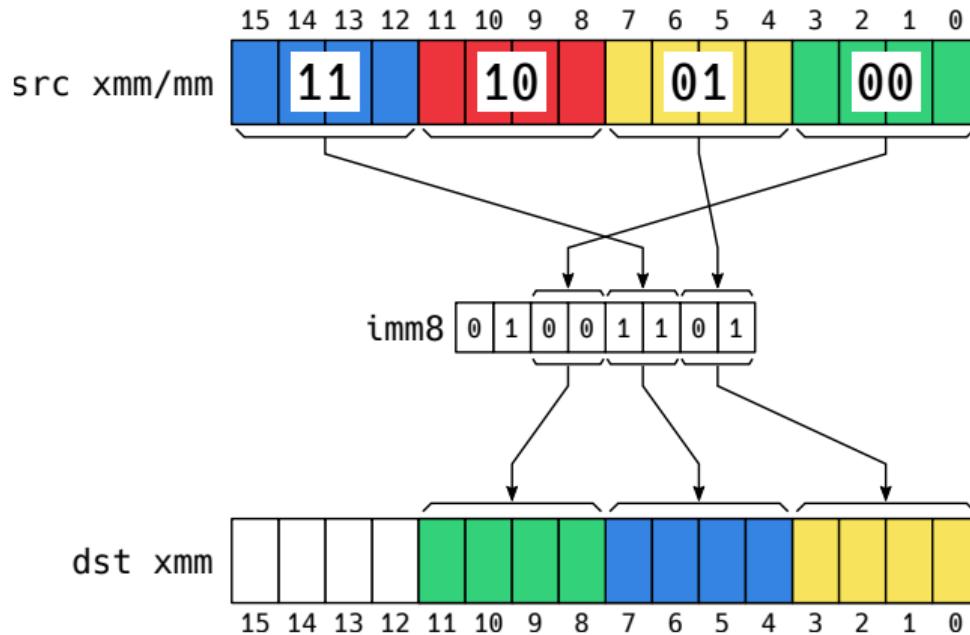
## Ejemplo - PSHUFD dst, src , imm8



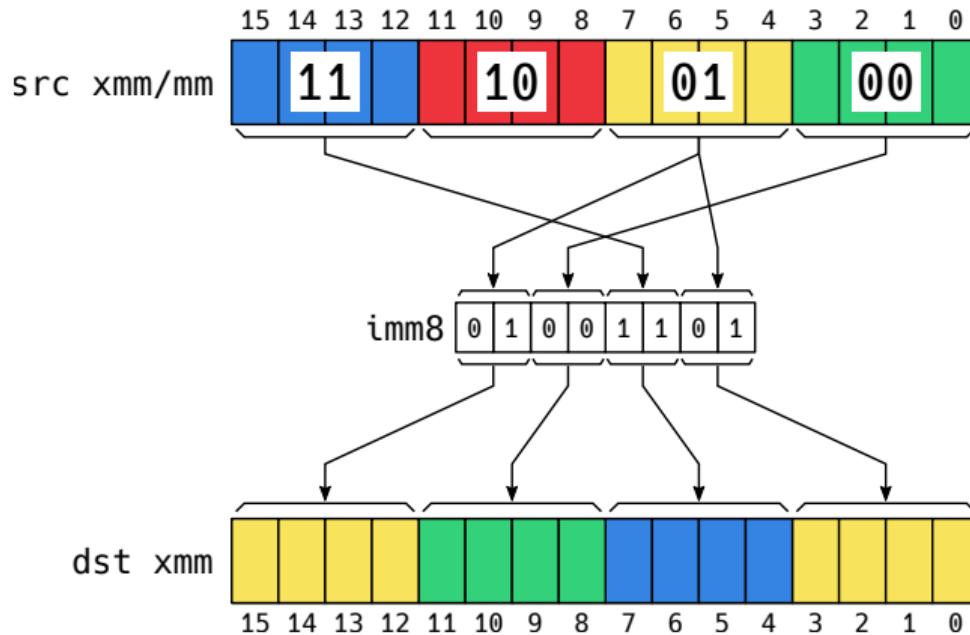
## Ejemplo - PSHUFD dst, src , imm8



## Ejemplo - PSHUFD dst, src , imm8



## Ejemplo - PSHUFD dst, src , imm8



# Shuffles

## SHUFPS—Shuffle Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF C6 /r ib SHUFPS <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE	Shuffle packed single-precision floating-point values selected by <i>imm8</i> from <i>xmm1</i> and <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.0F.WIG C6 /r ib VSHUFPS <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Shuffle Packed single-precision floating-point values selected by <i>imm8</i> from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.0F.WIG C6 /r ib VSHUFPS <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Shuffle Packed single-precision floating-point values selected by <i>imm8</i> from <i>ymm2</i> and <i>ymm3/mem</i> .

# Shuffles

## SHUFPS—Shuffle Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF C6 /r ib SHUFPS <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE	Shuffle packed single-precision floating-point values selected by <i>imm8</i> from <i>xmm1</i> and <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.0F.WIG C6 /r ib VSHUFPS <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Shuffle Packed single-precision floating-point values selected by <i>imm8</i> from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.0F.WIG C6 /r ib VSHUFPS <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Shuffle Packed single-precision floating-point values selected by <i>imm8</i> from <i>ymm2</i> and <i>ymm3/mem</i> .

### SHUFPS (128-bit Legacy SSE version)

~~DEST~~  
 $DEST[31:0] \leftarrow Select4(SRC1[127:0], imm8[1:0]);$   
 $DEST[63:32] \leftarrow Select4(SRC1[127:0], imm8[3:2]);$   
 $DEST[95:64] \leftarrow Select4(SRC1[127:0], imm8[5:4]);$   
 $DEST[127:96] \leftarrow Select4(SRC1[127:0], imm8[7:6]);$   
 $DEST[VLMAX-1:128] \text{ (Unmodified)}$

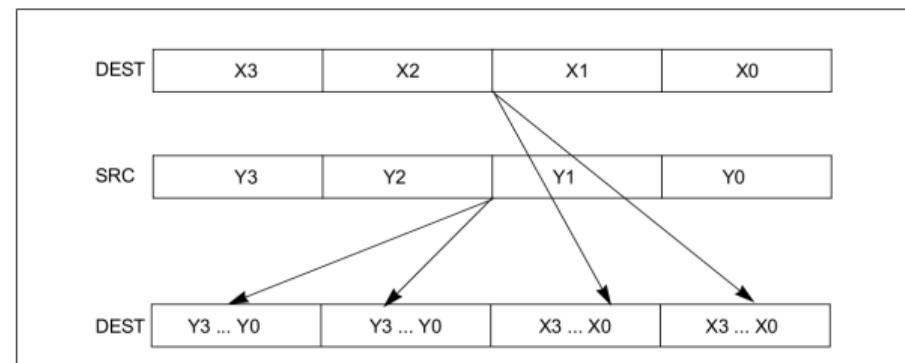
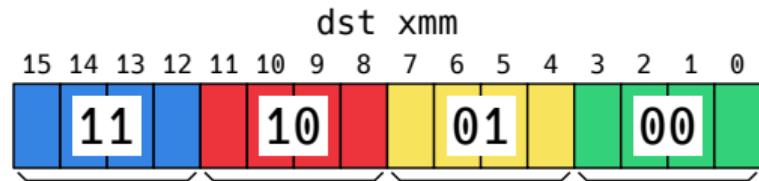
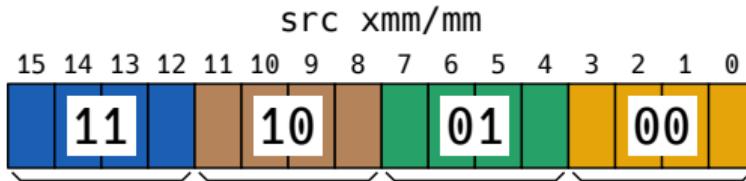
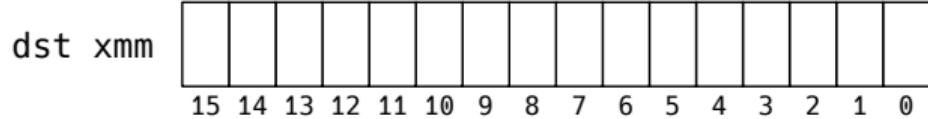


Figure 4-22. SHUFPS Shuffle Operation

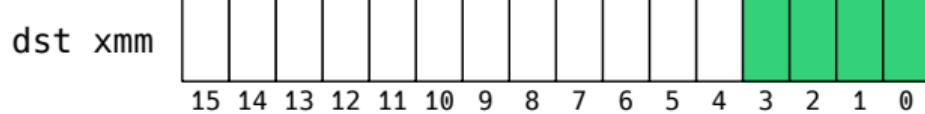
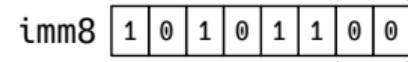
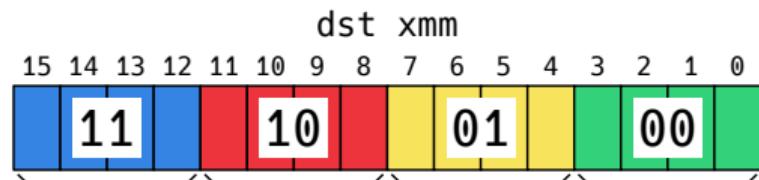
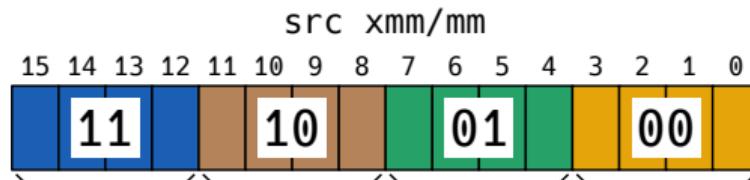
## Ejemplo - SHUFPS dst, src , imm8



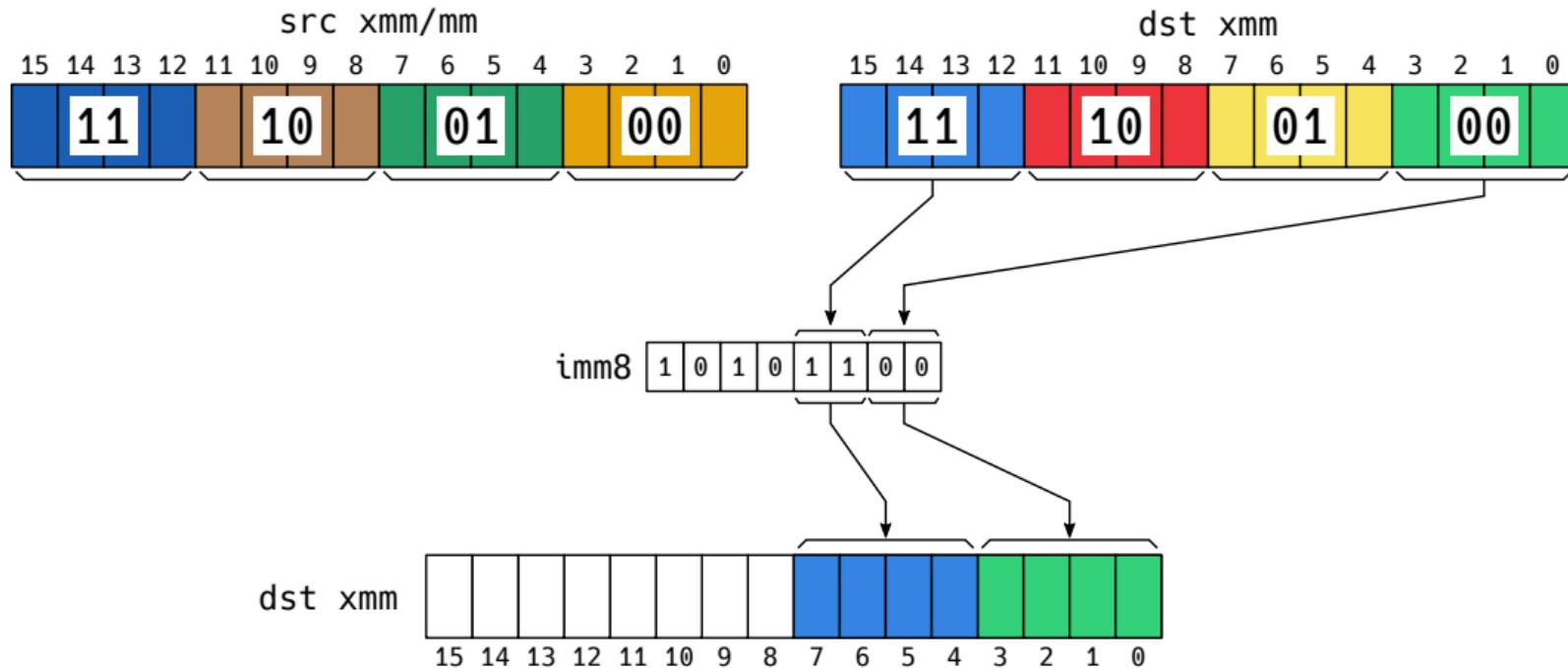
imm8 [ 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 ]



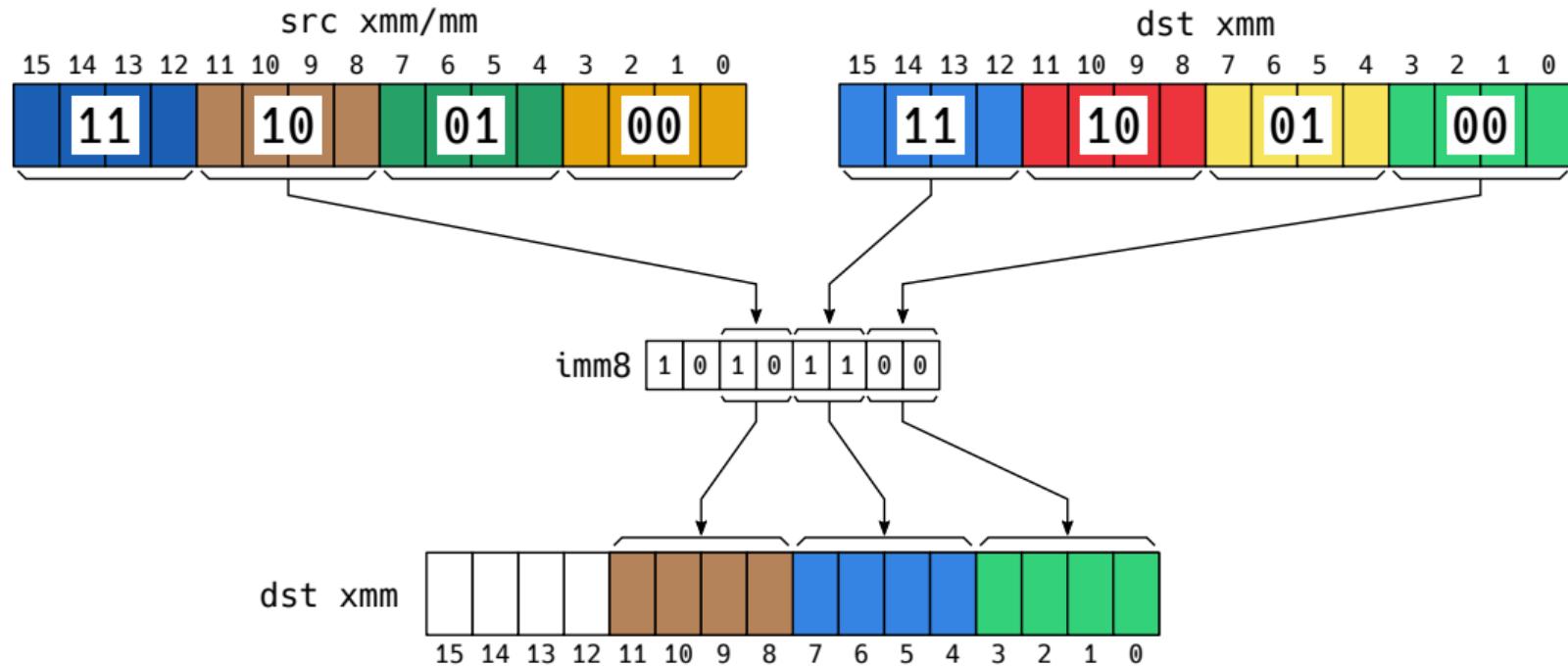
## Ejemplo - SHUFPS dst, src , imm8



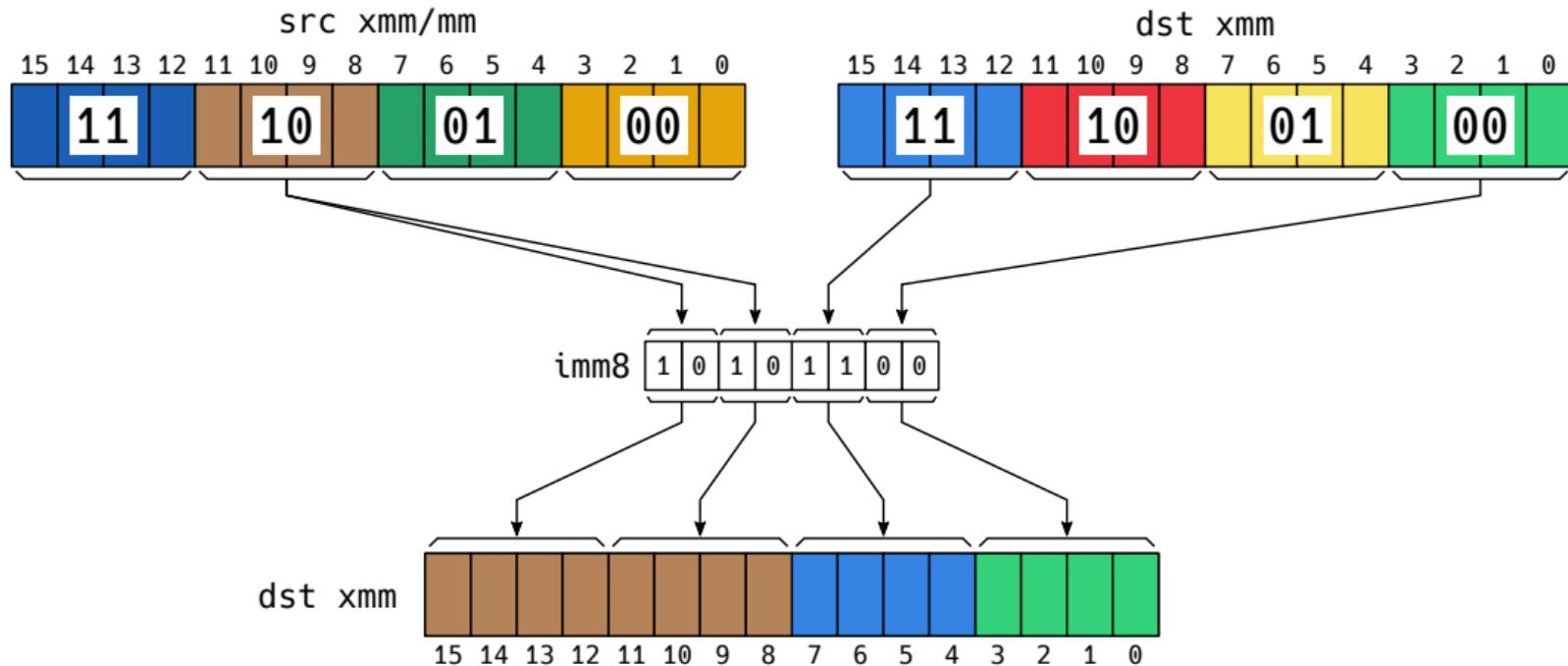
## Ejemplo - SHUFPS dst, src , imm8



## Ejemplo - SHUFPS dst, src , imm8



## Ejemplo - SHUFPS dst, src , imm8



# Shuffles

## SHUFPD—Shuffle Packed Double-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F C6 /rib SHUFPD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE2	Shuffle packed double-precision floating-point values selected by <i>imm8</i> from <i>xmm1</i> and <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG C6 /r ib VSHUFPD <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Shuffle Packed double-precision floating-point values selected by <i>imm8</i> from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG C6 /r ib VSHUFPD <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Shuffle Packed double-precision floating-point values selected by <i>imm8</i> from <i>ymm2</i> and <i>ymm3/mem</i> .

### SHUFPD (128-bit Legacy SSE version)

```

IF IMMO[0] = 0          DEST
    THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST[63:0] ← SRC1[127:64] Fl;
IF IMMO[1] = 0
    THEN DEST[127:64] ← SRC2[63:0]
    ELSE DEST[127:64] ← SRC2[127:64] Fl;
DEST[VLMAX-1:128] (Unmodified)
  
```

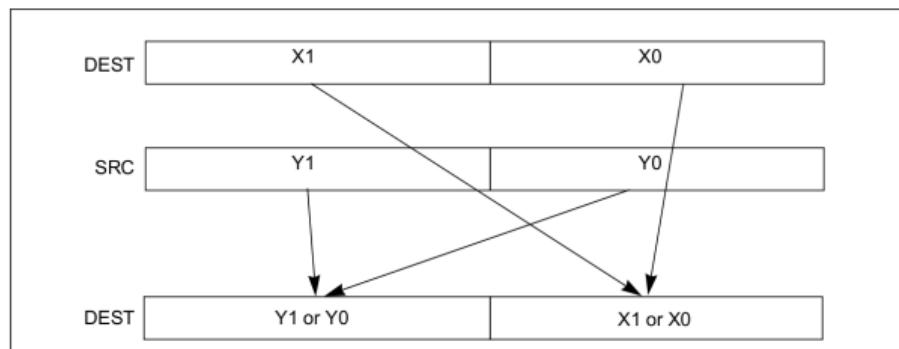
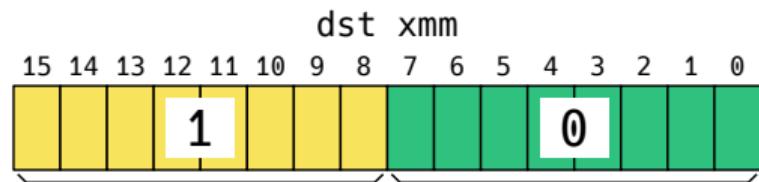
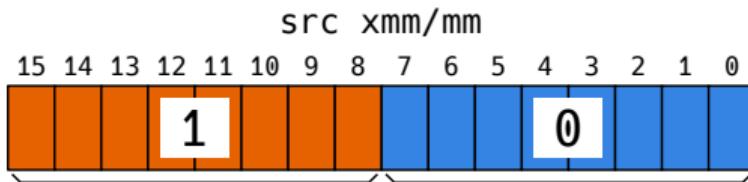
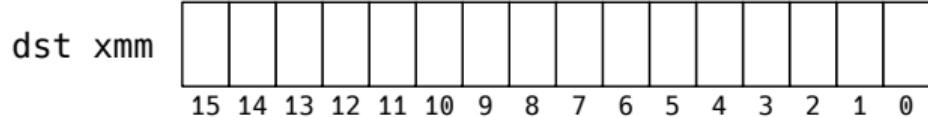


Figure 4-21. SHUFPD Shuffle Operation

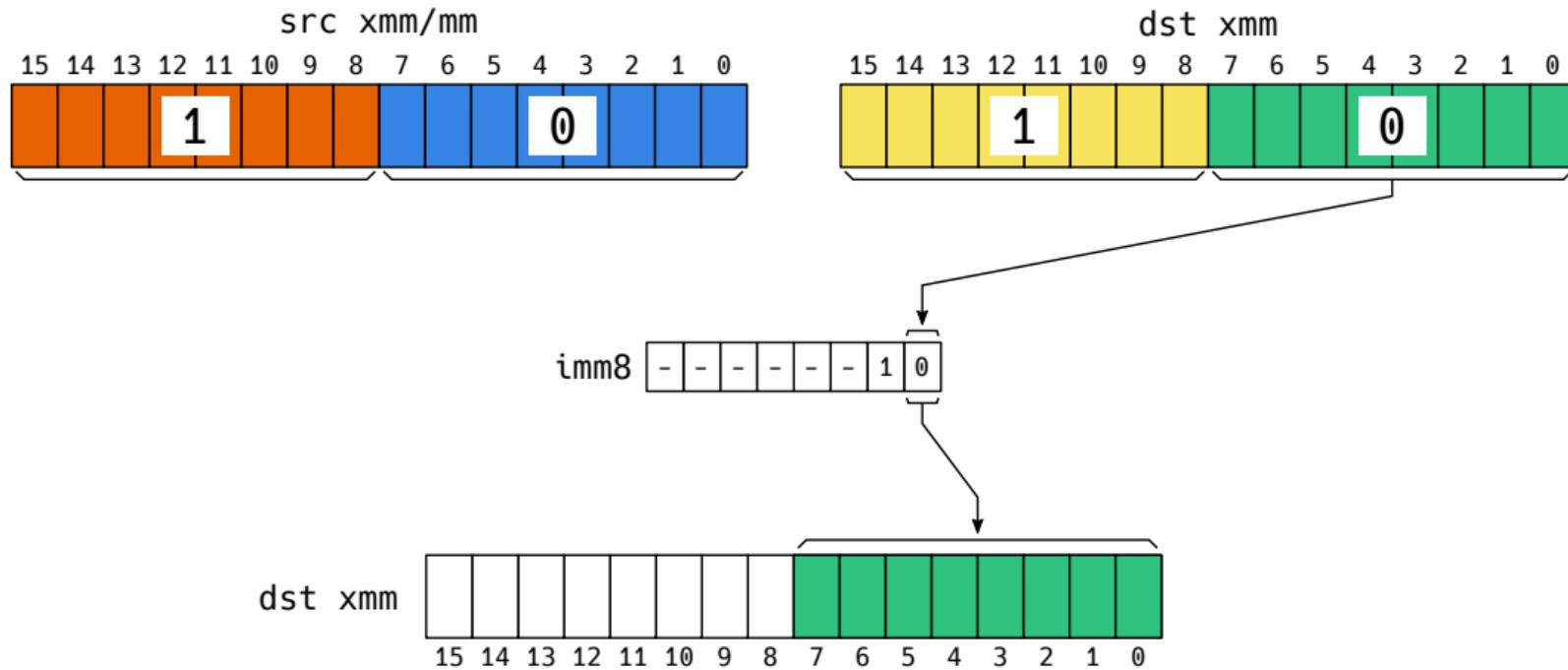
## Ejemplo - SHUFPD dst, src , imm8



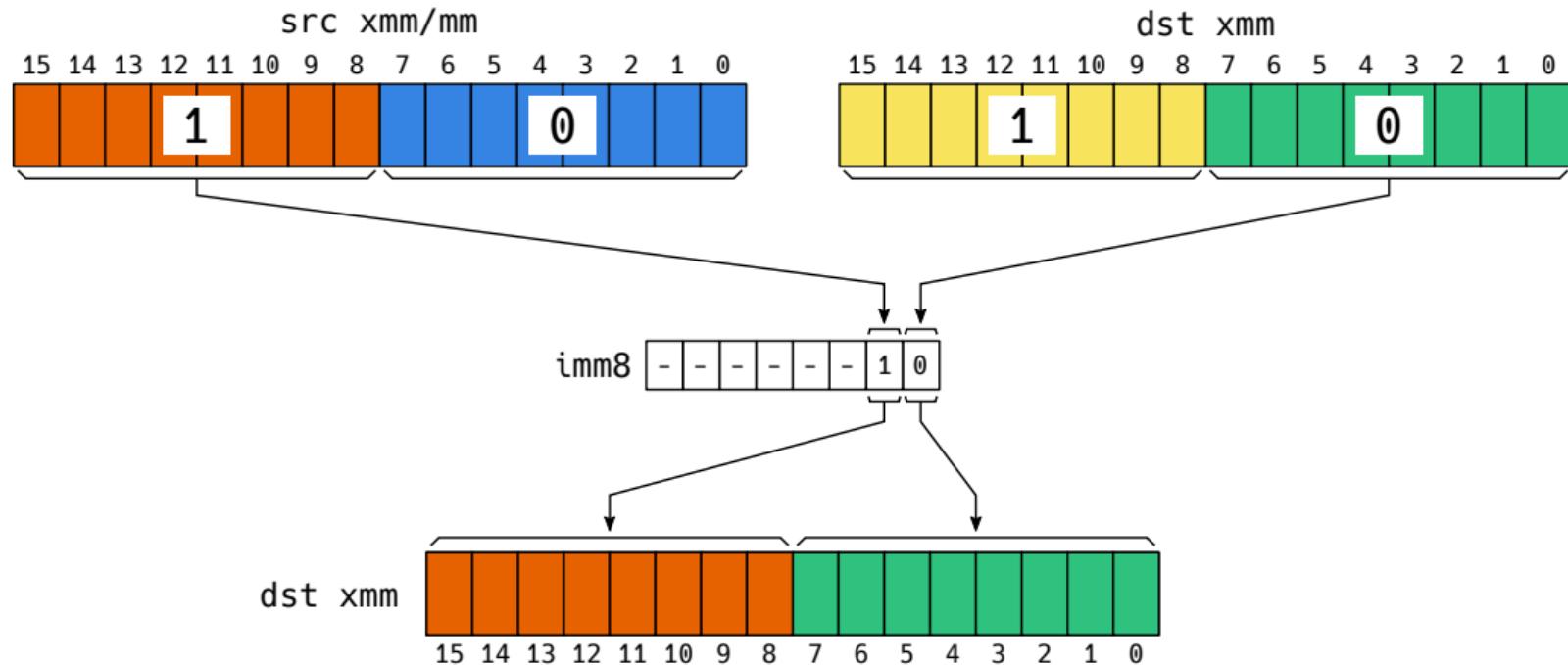
imm8 [- - - - - - 1 0]



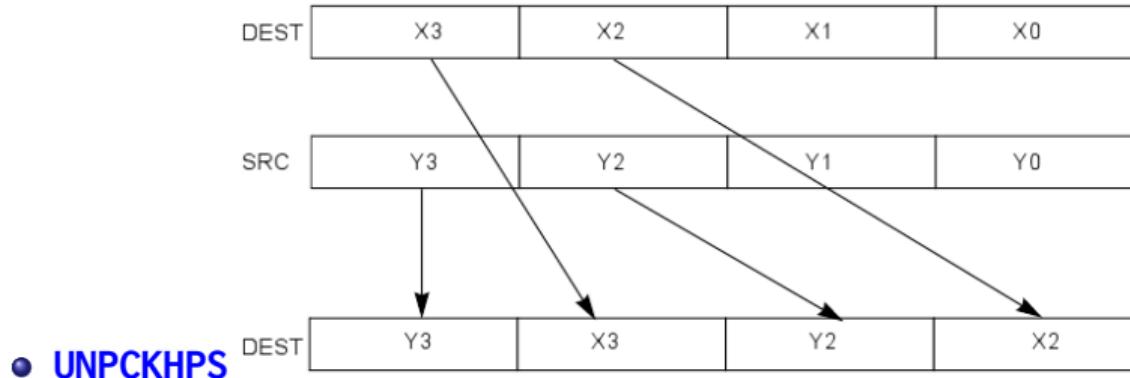
Ejemplo - SHUFPD dst, src , imm8



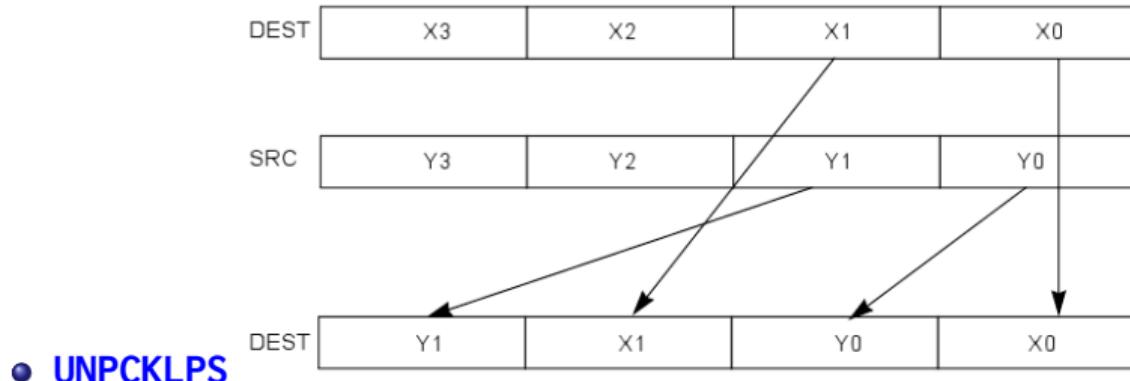
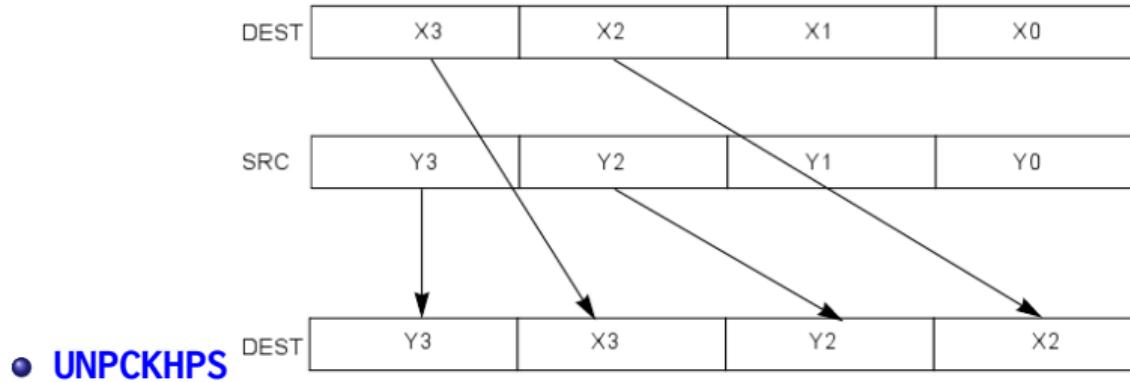
## Ejemplo - SHUFPD dst, src , imm8



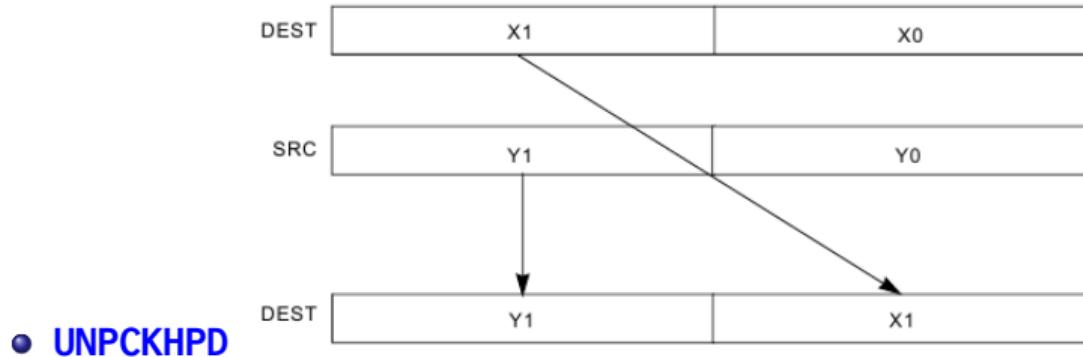
# Instrucciones de Shuffle y Comparación



# Instrucciones de Shuffle y Comparación



# Instrucciones de Shuffle y Comparación



● **UNPCKHPD**

# Instrucciones de Shuffle y Comparación



● UNPCKHPD



● UNPCKLPD

# Insert/Extract

Las instrucciones de *Insert* y *Extract*, permiten como su nombre lo indica, **insertar** y **extraer** valores dentro de un registro.

# Insert/Extract

Las instrucciones de *Insert* y *Extract*, permiten como su nombre lo indica, **insertar** y **extraer** valores dentro de un registro.

- **INSERTPS** - Insert Packed Single FP Value
- **EXTRACTPS** - Extract Packed Single FP Value
- **PINSRB** - Insert Byte
- **PINSRW** - Insert Word
- **PINSRD** - Insert Dword
- **PINSRQ** - Insert Qword
- **PEXTRB** - Extract Byte
- **PEXTRW** - Extract Word
- **PEXTRD** - Extract Dword
- **PEXTRQ** - Extract Qword

# Insert / Extract

## INSERTPS – Insert Packed Single Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 21 /r ib INSERTPS <i>xmm1, xmm2/m32, imm8</i>	RMI	V/V	SSE4_1	Insert a single precision floating-point value selected by <i>imm8</i> from <i>xmm2/m32</i> into <i>xmm1</i> at the specified destination element specified by <i>imm8</i> and zero out destination elements in <i>xmm1</i> as indicated in <i>imm8</i> .
VEX.NDS.128.66.0F3A.WIG 21 /r ib VINSERTPS <i>xmm1, xmm2, xmm3/m32, imm8</i>	RVMI	V/V	AVX	Insert a single precision floating point value selected by <i>imm8</i> from <i>xmm3/m32</i> and merge into <i>xmm2</i> at the specified destination element specified by <i>imm8</i> and zero out destination elements in <i>xmm1</i> as indicated in <i>imm8</i> .

# Insert / Extract

## INSERTPS – Insert Packed Single Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 21 /r ib INSERTPS <i>xmm1, xmm2/m32, imm8</i>	RMI	V/V	SSE4_1	Insert a single precision floating-point value selected by <i>imm8</i> from <i>xmm2/m32</i> into <i>xmm1</i> at the specified destination element specified by <i>imm8</i> and zero out destination elements in <i>xmm1</i> as indicated in <i>imm8</i> .
VEX.NDS.128.66.0F3A.WIG 21 /r ib VINSERTPS <i>xmm1, xmm2, xmm3/m32, imm8</i>	RVMI	V/V	AVX	Insert a single precision floating point value selected by <i>imm8</i> from <i>xmm3/m32</i> and merge into <i>xmm2</i> at the specified destination element specified by <i>imm8</i> and zero out destination elements in <i>xmm1</i> as indicated in <i>imm8</i> .

### INSERTPS (128-bit Legacy SSE version)

IF (SRC = REG) THEN COUNT\_S ← imm8[7:6]

ELSE COUNT\_S ← 0

COUNT\_D ← imm8[5:4]

ZMASK ← imm8[3:0]

CASE (COUNT\_S) OF

0: TMP ← SRC[31:0]

1: TMP ← SRC[63:32]

2: TMP ← SRC[95:64]

3: TMP ← SRC[127:96]

ESAC;

CASE (COUNT\_D) OF

0: TMP2[31:0] ← TMP

TMP2[127:32] ← DEST[127:32]

1: TMP2[63:32] ← TMP

TMP2[31:0] ← DEST[31:0]

TMP2[127:64] ← DEST[127:64]

2: TMP2[95:64] ← TMP

TMP2[63:0] ← DEST[63:0]

TMP2[127:96] ← DEST[127:96]

3: TMP2[127:96] ← TMP

TMP2[95:0] ← DEST[95:0]

ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] ← 00000000H

ELSE DEST[31:0] ← TMP2[31:0]

IF (ZMASK[1] = 1) THEN DEST[63:32] ← 00000000H

ELSE DEST[63:32] ← TMP2[63:32]

IF (ZMASK[2] = 1) THEN DEST[95:64] ← 00000000H

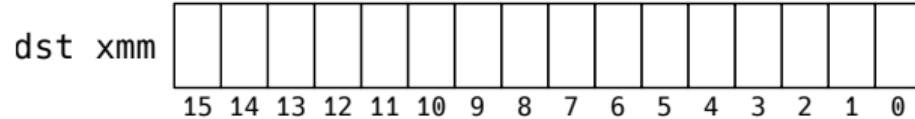
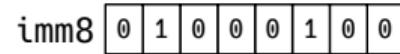
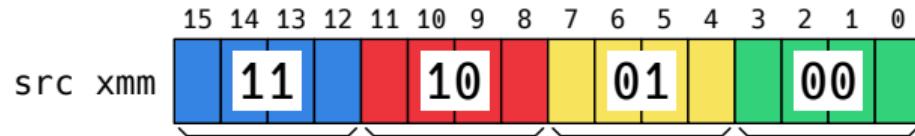
ELSE DEST[95:64] ← TMP2[95:64]

IF (ZMASK[3] = 1) THEN DEST[127:96] ← 00000000H

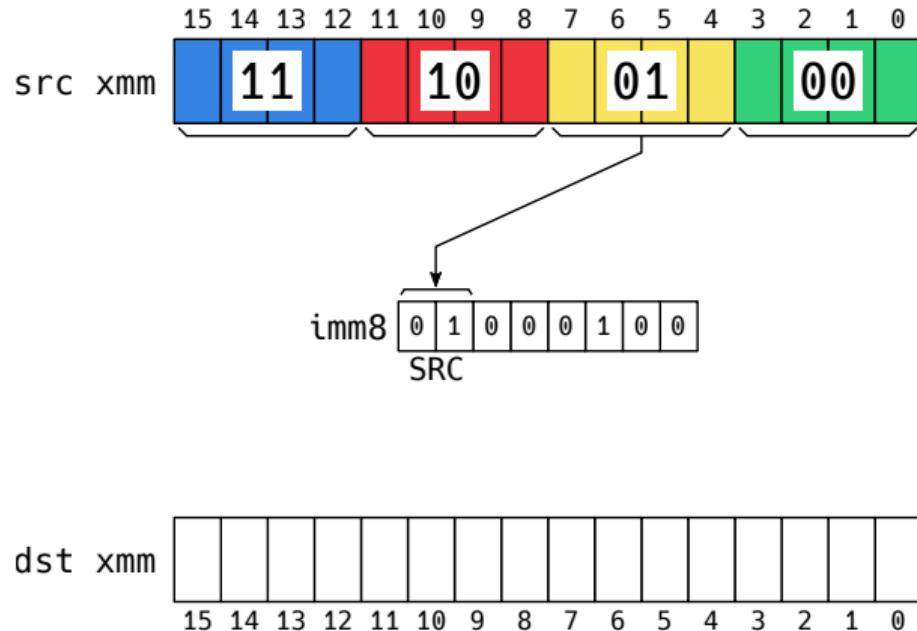
ELSE DEST[127:96] ← TMP2[127:96]

DEST[VLMAX-1:128] (Unmodified)

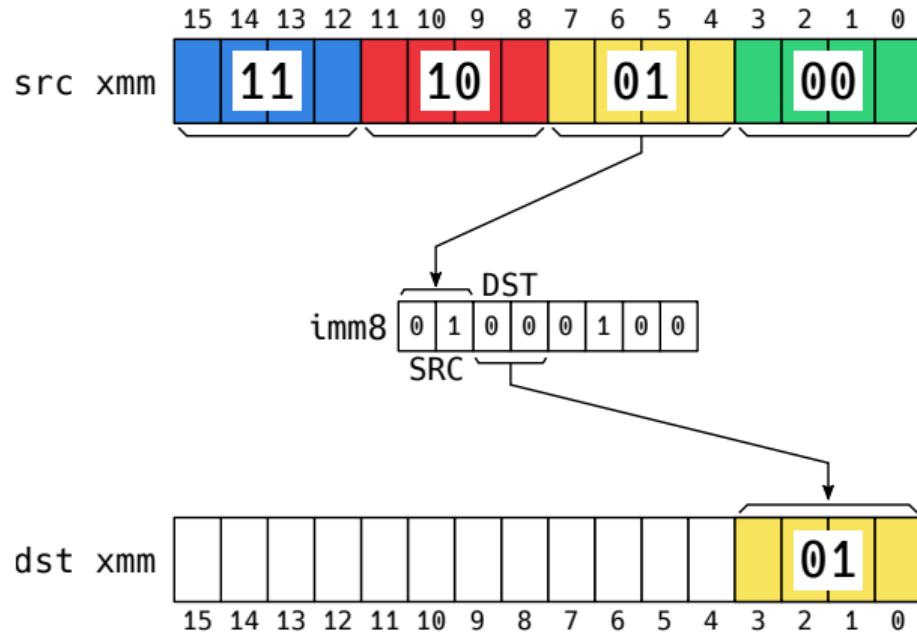
## Ejemplo - INSERTPS dst, src , imm8



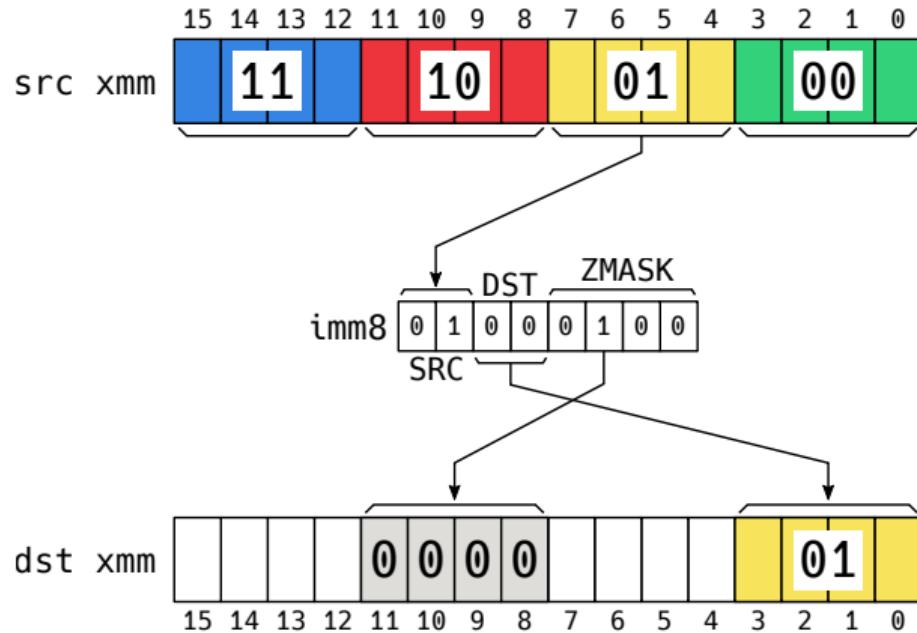
## Ejemplo - INSERTPS dst, src , imm8



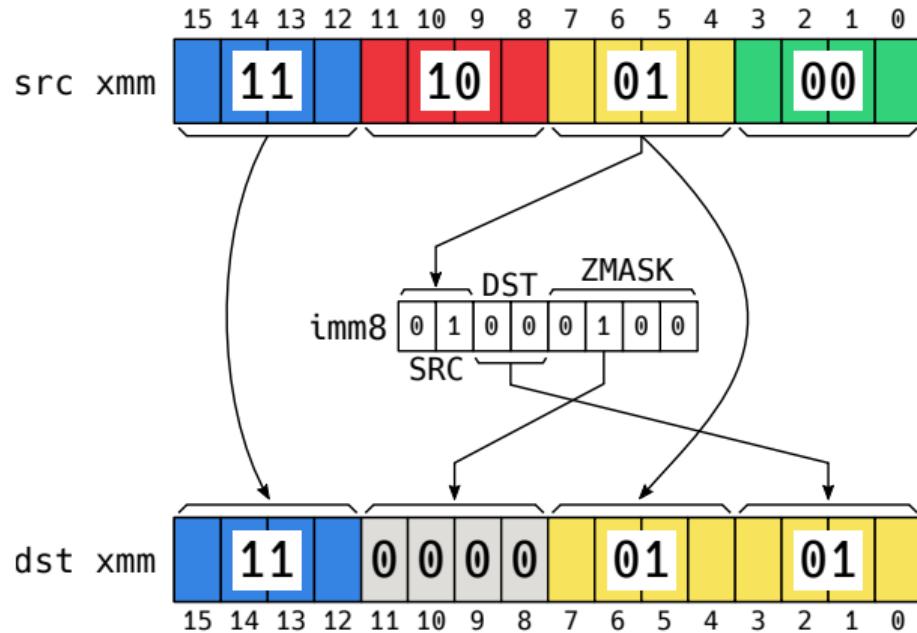
## Ejemplo - INSERTPS dst, src , imm8



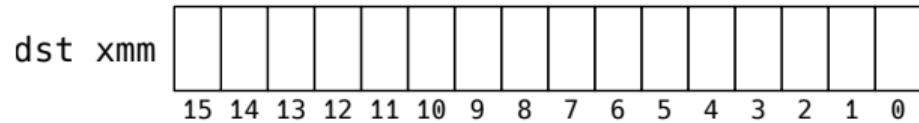
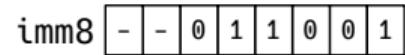
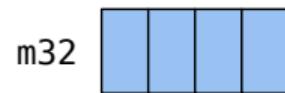
## Ejemplo - INSERTPS dst, src , imm8



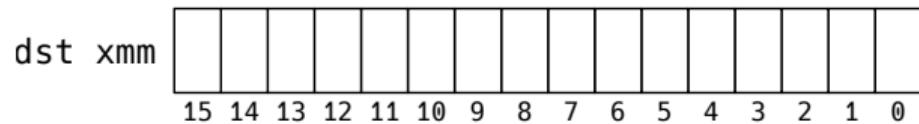
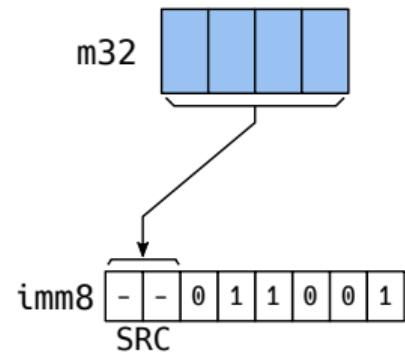
## Ejemplo - INSERTPS dst, src , imm8



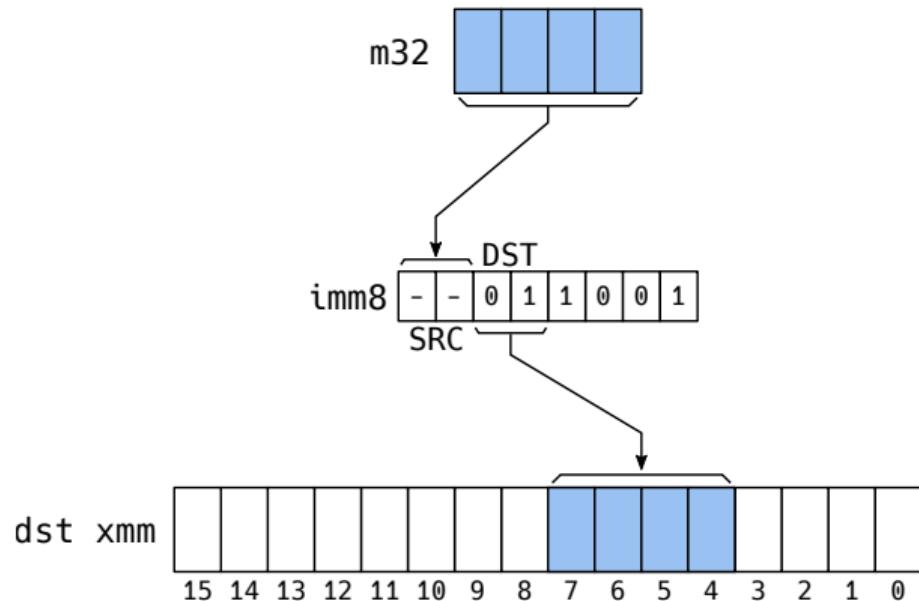
## Ejemplo - INSERTPS dst, src , imm8



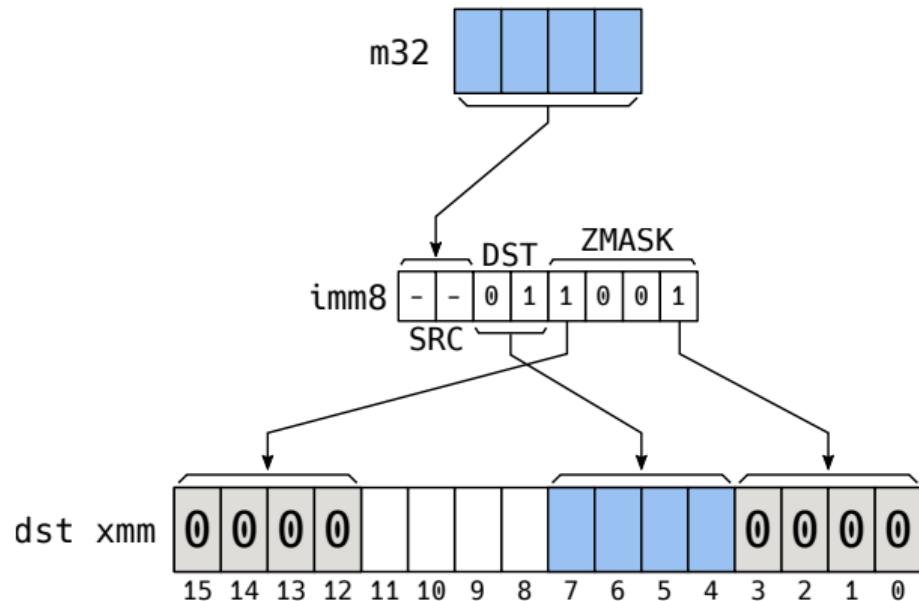
## Ejemplo - INSERTPS dst, src , imm8



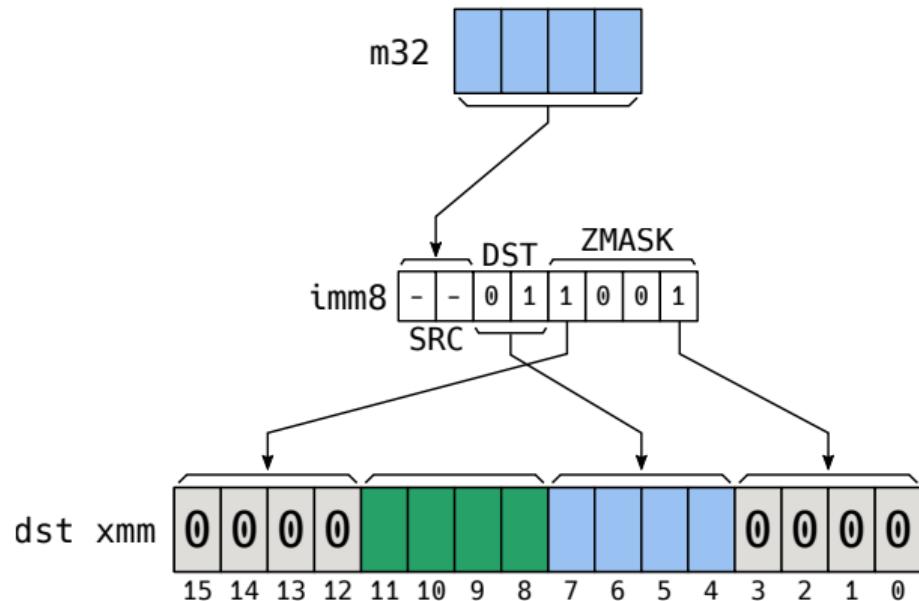
## Ejemplo - INSERTPS dst, src , imm8



## Ejemplo - INSERTPS dst, src , imm8



## Ejemplo - INSERTPS dst, src , imm8



# Insert / Extract

## EXTRACTPS – Extract Packed Single Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 17 <i>/r ib</i> EXTRACTPS <i>reg/m32, xmm2, imm8</i>	MRI	V/V	SSE4_1	Extract a single-precision floating-point value from <i>xmm2</i> at the source offset specified by <i>imm8</i> and store the result to <i>reg or m32</i> . The upper 32 bits of r64 is zeroed if reg is r64.
VEX.128.66.0F3A.WIG 17 <i>/r ib</i> VEXTRACTPS <i>r/m32, xmm1, imm8</i>	MRI	V/V	AVX	Extract one single-precision floating-point value from <i>xmm1</i> at the offset specified by <i>imm8</i> and store the result in <i>reg or m32</i> . Zero extend the results in 64-bit register if applicable.

# Insert / Extract

## EXTRACTPS – Extract Packed Single Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 17 /r ib EXTRACTPS <i>reg/m32, xmm2, imm8</i>	MRI	V/V	SSE4_1	Extract a single-precision floating-point value from <i>xmm2</i> at the source offset specified by <i>imm8</i> and store the result to <i>reg or m32</i> . The upper 32 bits of r64 is zeroed if reg is r64.
VEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS <i>r/m32, xmm1, imm8</i>	MRI	V/V	AVX	Extract one single-precision floating-point value from <i>xmm1</i> at the offset specified by <i>imm8</i> and store the result in <i>reg or m32</i> . Zero extend the results in 64-bit register if applicable.

### EXTRACTPS (128-bit Legacy SSE version)

*SRC\_OFFSET*  $\leftarrow$  IMM8[1:0]

IF ( 64-Bit Mode and DEST is register)

*DEST*[31:0]  $\leftarrow$  (*SRC*[127:0]  $\gg$  (*SRC\_OFFSET*\*32)) AND OFFFFFFFh

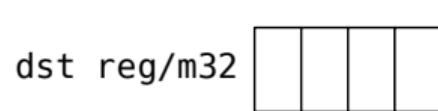
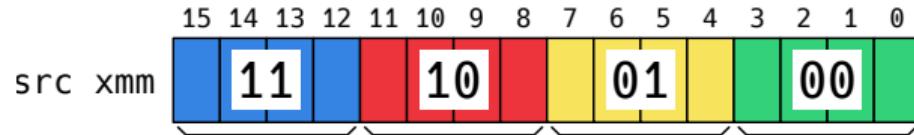
*DEST*[63:32]  $\leftarrow$  0

ELSE

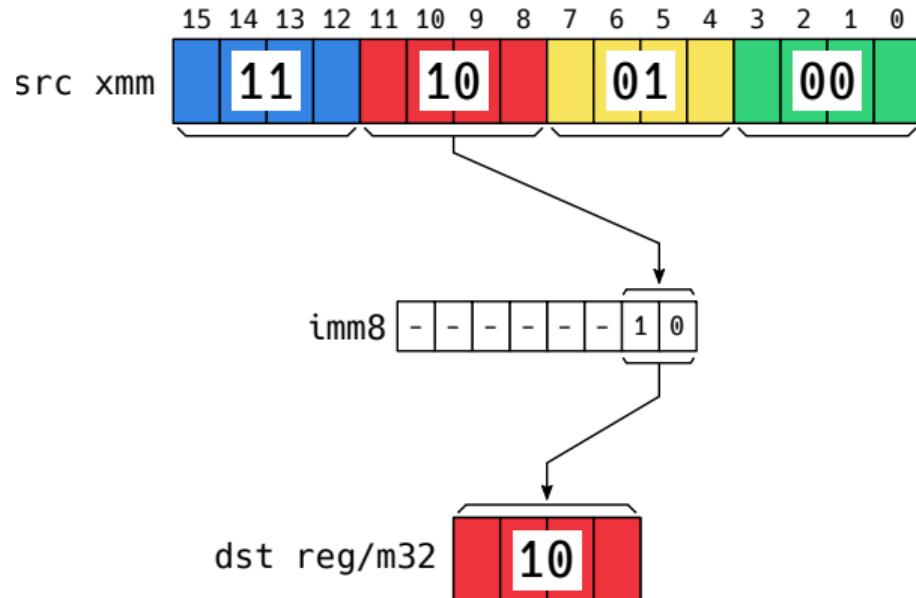
*DEST*[31:0]  $\leftarrow$  (*SRC*[127:0]  $\gg$  (*SRC\_OFFSET*\*32)) AND OFFFFFFFh

FI

## Ejemplo - EXTRACTPS dst, src , imm8



## Ejemplo - EXTRACTPS dst, src , imm8



# Insert / Extract

## PINSRB/PINSRD/PINSRQ – Insert Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 20 /r ib PINSRB <i>xmm1, r32/m8, imm8</i>	RMI	V/V	SSE4_1	Insert a byte integer value from <i>r32/m8</i> into <i>xmm1</i> at the destination element in <i>xmm1</i> specified by <i>imm8</i> .
66 0F 3A 22 /r ib PINSRD <i>xmm1, r/m32, imm8</i>	RMI	V/V	SSE4_1	Insert a dword integer value from <i>r/m32</i> into the <i>xmm1</i> at the destination element specified by <i>imm8</i> .
66 REX.W 0F 3A 22 /r ib PINSRQ <i>xmm1, r/m64, imm8</i>	RMI	V/N. E.	SSE4_1	Insert a qword integer value from <i>r/m64</i> into the <i>xmm1</i> at the destination element specified by <i>imm8</i> .
VEX.NDS.128.66.0F3A.W0 20 /r ib VPINSRB <i>xmm1, xmm2, r32/m8, imm8</i>	RVMI	V <sup>1</sup> /V	AVX	Merge a byte integer value from <i>r32/m8</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the byte offset in <i>imm8</i> .
VEX.NDS.128.66.0F3A.W0 22 /r ib VPINSRD <i>xmm1, xmm2, r/m32, imm8</i>	RVMI	V/V	AVX	Insert a dword integer value from <i>r32/m32</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the dword offset in <i>imm8</i> .
VEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRQ <i>xmm1, xmm2, r/m64, imm8</i>	RVMI	V/I	AVX	Insert a qword integer value from <i>r64/m64</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the qword offset in <i>imm8</i> .

# Insert / Extract

## PINSRB/PINSRD/PINSRQ – Insert Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 20 /r ib PINSRB <i>xmm1, r32/m8, imm8</i>	RMI	V/V	SSE4_1	Insert a byte integer value from <i>r32/m8</i> into <i>xmm1</i> at the destination element in <i>xmm1</i> specified by <i>imm8</i> .
66 0F 3A 22 /r ib PINSRD <i>xmm1, r/m32, imm8</i>	RMI	V/V	SSE4_1	Insert a dword integer value from <i>r/m32</i> into the <i>xmm1</i> at the destination element specified by <i>imm8</i> .
66 REX.W 0F 3A 22 /r ib PINSRQ <i>xmm1, r/m64, imm8</i>	RMI	V/N.		
VEX.NDS.128.66.0F3A.W0 20 /r ib VPINSRB <i>xmm1, xmm2, r32/m8, imm8</i>	RVMI	V <sup>1</sup> /V		<p>CASE OF</p> <p>PINSRB: SEL <math>\leftarrow</math> COUNT[3:0];      MASK <math>\leftarrow</math> (0FFH <math>\ll</math> (SEL * 8));      TEMP <math>\leftarrow</math> (((SRC[7:0] <math>\ll</math> (SEL * 8)) AND MASK);</p> <p>PINSRD: SEL <math>\leftarrow</math> COUNT[1:0];      MASK <math>\leftarrow</math> (0FFFFFFFH <math>\ll</math> (SEL * 32));      TEMP <math>\leftarrow</math> (((SRC <math>\ll</math> (SEL * 32)) AND MASK) ;</p> <p>PINSRQ: SEL <math>\leftarrow</math> COUNT[0]      MASK <math>\leftarrow</math> (0xFFFFFFFFFFFFFFFH <math>\ll</math> (SEL * 64));      TEMP <math>\leftarrow</math> (((SRC <math>\ll</math> (SEL * 32)) AND MASK) ;</p> <p>ESAC;      DEST <math>\leftarrow</math> ((DEST AND NOT MASK) OR TEMP);</p>
VEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRQ <i>xmm1, xmm2, r/m64, imm8</i>	RVMI	V/I		

# Insert / Extract

## PINSRW—Insert Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF C4 /r ib <sup>1</sup> PINSRW <i>mm, r32/m16, imm8</i>	RMI	V/V	SSE	Insert the low word from <i>r32</i> or from <i>m16</i> into <i>mm</i> at the word position specified by <i>imm8</i> .
66 OF C4 /r ib PINSRW <i>xmm, r32/m16, imm8</i>	RMI	V/V	SSE2	Move the low word of <i>r32</i> or from <i>m16</i> into <i>xmm</i> at the word position specified by <i>imm8</i> .
VEX.NDS.128.66.0F.W0 C4 /r ib VPINSRW <i>xmm1, xmm2, r32/m16, imm8</i>	RVMI	V <sup>2</sup> /V	AVX	Insert a word integer value from <i>r32/m16</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the word offset in <i>imm8</i> .

# Insert / Extract

## PINSRW—Insert Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF C4 /r ib <sup>1</sup> PINSRW <i>mm, r32/m16, imm8</i>	RMI	V/V	SSE	Insert the low word from <i>r32</i> or from <i>m16</i> into <i>mm</i> at the word position specified by <i>imm8</i> .
66 OF C4 /r ib PINSRW <i>xmm, r32/m16, imm8</i>	RMI	V/V	SSE2	Move the low word of <i>r32</i> or from <i>m16</i> into <i>xmm</i> at the word position specified by <i>imm8</i> .
VEX.NDS.128.66.0F.W0 C4 /r ib VPINSRW <i>xmm1, xmm2, r32/m16, imm8</i>	RVMI	V <sup>2</sup> /V	AVX	Insert a word integer value from <i>r32/m16</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the word offset in <i>imm8</i> .

### PINSRW (with 128-bit source operand)

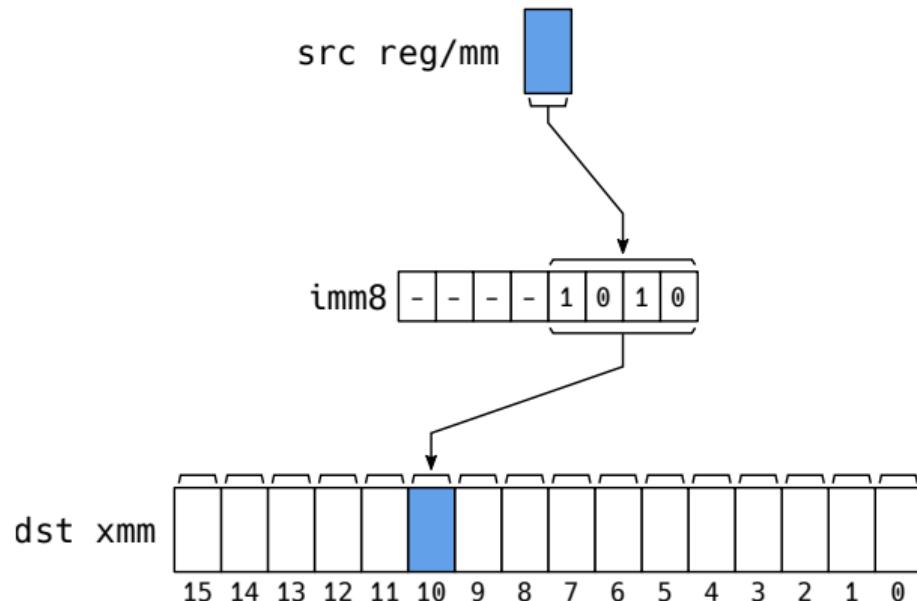
SEL  $\leftarrow$  COUNT AND 7H;

CASE (Determine word position) OF

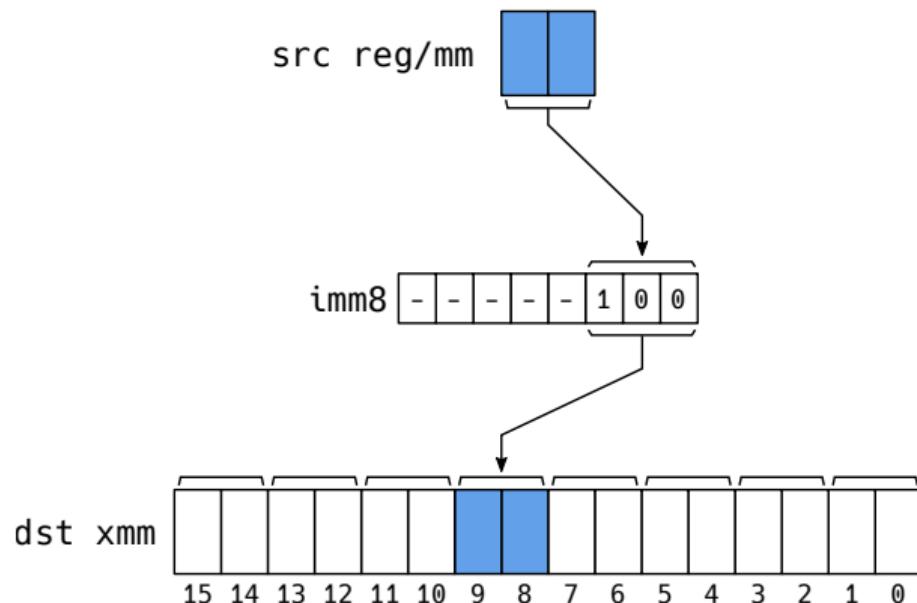
- SEL  $\leftarrow$  0: MASK  $\leftarrow$  0000000000000000000000000000FFFFH;
- SEL  $\leftarrow$  1: MASK  $\leftarrow$  0000000000000000000000000000FFFOOOOH;
- SEL  $\leftarrow$  2: MASK  $\leftarrow$  0000000000000000000000000000FFFFFO0000000OH;
- SEL  $\leftarrow$  3: MASK  $\leftarrow$  0000000000000000000000000000FFFFFO000000000000OH;
- SEL  $\leftarrow$  4: MASK  $\leftarrow$  000000000000FFFFFO0000000000000000000000OH;
- SEL  $\leftarrow$  5: MASK  $\leftarrow$  00000000FFFFFO000000000000000000000000OH;
- SEL  $\leftarrow$  6: MASK  $\leftarrow$  0000FFFFFO00000000000000000000000000000000OH;
- SEL  $\leftarrow$  7: MASK  $\leftarrow$  FFFF00000000000000000000000000000000000000OH;

DEST  $\leftarrow$  (DEST AND NOT MASK) OR (((SRC  $\ll$  (SEL \* 16)) AND MASK);

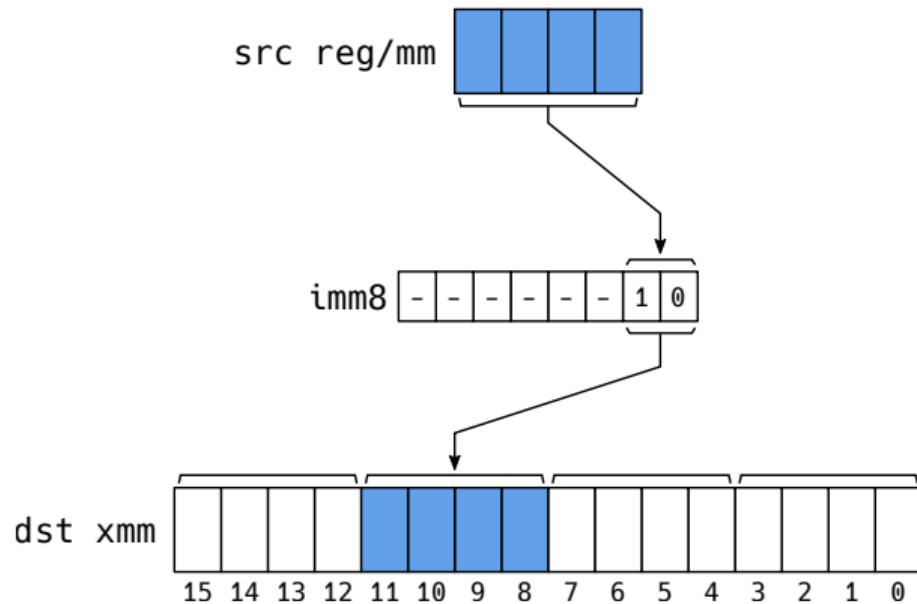
## Ejemplo - PINSRB dst, src , imm8



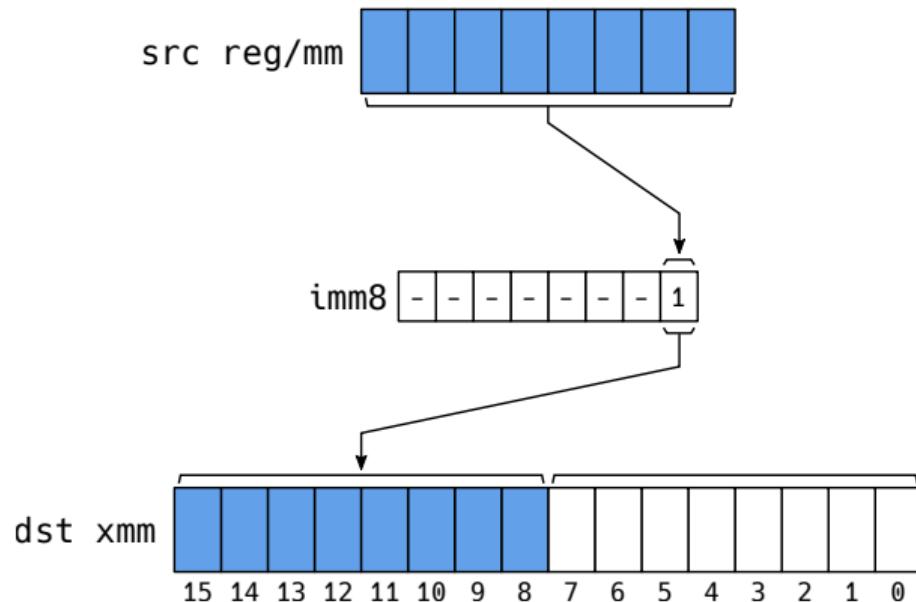
## Ejemplo - PINSRW dst, src , imm8



## Ejemplo - PINSRD dst, src , imm8



## Ejemplo - PINSRQ dst, src , imm8



# Insert / Extract

## PEXTRB/PEXTRD/PEXTRQ – Extract Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 OF 3A 14 /r ib PEXTRB <i>reg/m8, xmm2, imm8</i>	MRI	V/V	SSE4_1	Extract a byte integer value from <i>xmm2</i> at the source byte offset specified by <i>imm8</i> into <i>reg</i> or <i>m8</i> . The upper bits of r32 or r64 are zeroed.
66 OF 3A 16 /r ib PEXTRD <i>r/m32, xmm2, imm8</i>	MRI	V/V	SSE4_1	Extract a dword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r/m32</i> .
66 REX.W OF 3A 16 /r ib PEXTRQ <i>r/m64, xmm2, imm8</i>	MRI	V/N.E.	SSE4_1	Extract a qword integer value from <i>xmm2</i> at the source qword offset specified by <i>imm8</i> into <i>r/m64</i> .
VEX.128.66.0F3A.W0 14 /r ib VPEXTRB <i>reg/m8, xmm2, imm8</i>	MRI	V <sup>1</sup> /V	AVX	Extract a byte integer value from <i>xmm2</i> at the source byte offset specified by <i>imm8</i> into <i>reg</i> or <i>m8</i> . The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 16 /r ib VPEXTRD <i>r32/m32, xmm2, imm8</i>	MRI	V/V	AVX	Extract a dword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r32/m32</i> .
VEX.128.66.0F3A.W1 16 /r ib VPEXTRQ <i>r64/m64, xmm2, imm8</i>	MRI	V/i	AVX	Extract a qword integer value from <i>xmm2</i> at the source qword offset specified by <i>imm8</i> into <i>r64/m64</i> .

# Insert / Extract

## PEXTRB/PEXTRD/PEXTRQ – Extract Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode	CPUID Support	Description
66 0F 3A 14 /r ib PEXTRB <i>reg/m8, xmm2, imm8</i>	MRI			<p>CASE of</p> <pre> PEXTRB: SEL ← COUNT[3:0]; TEMP ← (Src &gt;&gt; SEL*8) AND FFH; IF (DEST = Mem8)     THEN         Mem8 ← TEMP[7:0]; ELSE IF (64-Bit Mode and 64-bit register selected)     THEN         R64[7:0] ← TEMP[7:0];         r64[63:8] ← ZERO_FILL; }; ELSE     R32[7:0] ← TEMP[7:0];     r32[31:8] ← ZERO_FILL; }; FI; </pre>
66 0F 3A 16 /r ib PEXTRD <i>r/m32, xmm2, imm8</i>	MRI			<pre> PEXTRD:SEL ← COUNT[1:0]; TEMP ← (Src &gt;&gt; SEL*32) AND FFFF_FFFFH; DEST ← TEMP; </pre>
66 REX.W 0F 3A 16 /r ib PEXTRQ <i>r/m64, xmm2, imm8</i>	MRI			<pre> PEXTRQ: SEL ← COUNT[0]; TEMP ← (Src &gt;&gt; SEL*64); DEST ← TEMP; </pre>
VEX.128.66.0F3A.W0 14 /r ib VPEXTRB <i>reg/m8, xmm2, imm8</i>	MRI			EASC:
VEX.128.66.0F3A.W0 16 /r ib VPEXTRD <i>r32/m32, xmm2, imm8</i>	MRI			
VEX.128.66.0F3A.W1 16 /r ib VPEXTRQ <i>r64/m64, xmm2, imm8</i>	MRI			

# Insert / Extract

## PEXTRW—Extract Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF C5 /r ib <sup>1</sup> PEXTRW <i>reg, mm, imm8</i>	RMI	V/V	SSE	Extract the word specified by <i>imm8</i> from <i>mm</i> and move it to <i>reg</i> , bits 15:0. The upper bits of r32 or r64 is zeroed.
66 OF C5 /r ib PEXTRW <i>reg, xmm, imm8</i>	RMI	V/V	SSE2	Extract the word specified by <i>imm8</i> from <i>xmm</i> and move it to <i>reg</i> , bits 15:0. The upper bits of r32 or r64 is zeroed.
66 OF 3A 15 /r ib PEXTRW <i>reg/m16, xmm, imm8</i>	MRI	V/V	SSE4_1	Extract the word specified by <i>imm8</i> from <i>xmm</i> and copy it to lowest 16 bits of <i>reg</i> or <i>m16</i> . Zero-extend the result in the destination, r32 or r64.
VEX.128.66.0F.W0 C5 /r ib VPEXTRW <i>reg, xmm1, imm8</i>	RMI	V <sup>2</sup> /V	AVX	Extract the word specified by <i>imm8</i> from <i>xmm1</i> and move it to <i>reg</i> , bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 15 /r ib VPEXTRW <i>reg/m16, xmm2, imm8</i>	MRI	V/V	AVX	Extract a word integer value from <i>xmm2</i> at the source word offset specified by <i>imm8</i> into <i>reg</i> or <i>m16</i> . The upper bits of r64/r32 is filled with zeros.

# Insert / Extract

## PEXTRW—Extract Word

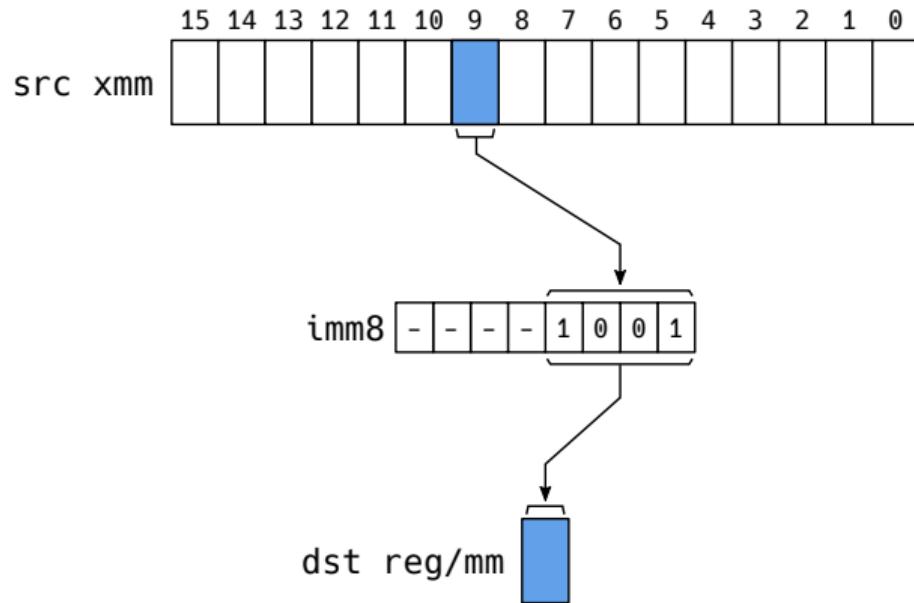
Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF C5 /rib <sup>1</sup> PEXTRW <i>reg, mm, imm8</i>	RMI	V/V	SSE	Extract the word specified by <i>imm8</i> from <i>mm</i> and move it to <i>reg</i> , bits 15:0. The upper bits of <i>r32</i> or <i>r64</i> are zeroed.

```

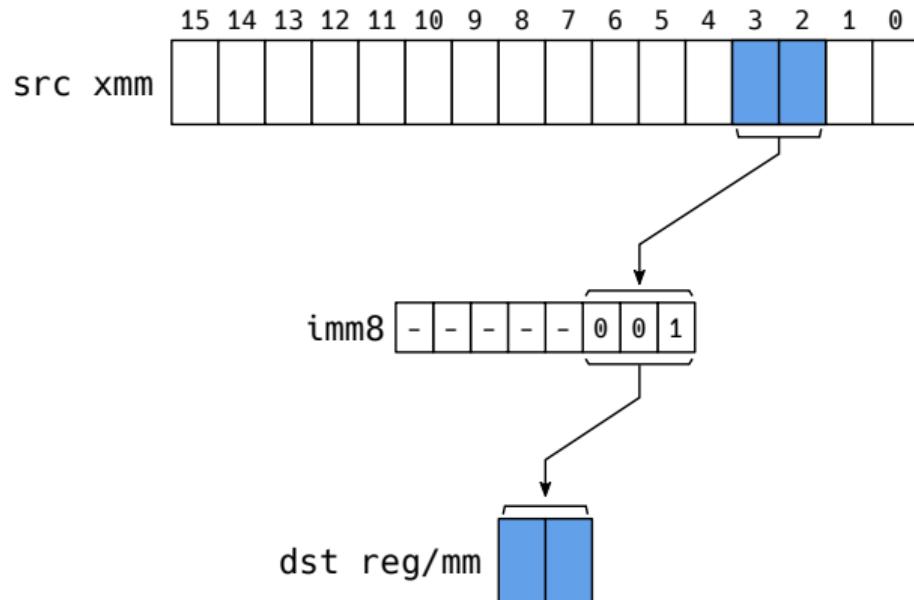
IF (DEST = Mem16)
THEN
    SEL ← COUNT[2:0];
    TEMP ← (Src >> SEL*16) AND FFFFH;
    Mem16 ← TEMP[15:0];
ELSE IF (64-Bit Mode and destination is a general-purpose register)
THEN
    FOR (PEXTRW instruction with 64-bit source operand)
        { SEL ← COUNT[1:0];
          TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
          r64[15:0] ← TEMP[15:0];
          r64[63:16] ← ZERO_FILL; };
    FOR (PEXTRW instruction with 128-bit source operand)
        { SEL ← COUNT[2:0];
          TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
          r64[15:0] ← TEMP[15:0];
          r64[63:16] ← ZERO_FILL; }
    ELSE
        FOR (PEXTRW instruction with 64-bit source operand)
            { SEL ← COUNT[1:0];
              TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
              r32[15:0] ← TEMP[15:0];
              r32[31:16] ← ZERO_FILL; };
        FOR (PEXTRW instruction with 128-bit source operand)
            { SEL ← COUNT[2:0];
              TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
              r32[15:0] ← TEMP[15:0];
              r32[31:16] ← ZERO_FILL; };
    Fl;
Fl;

```

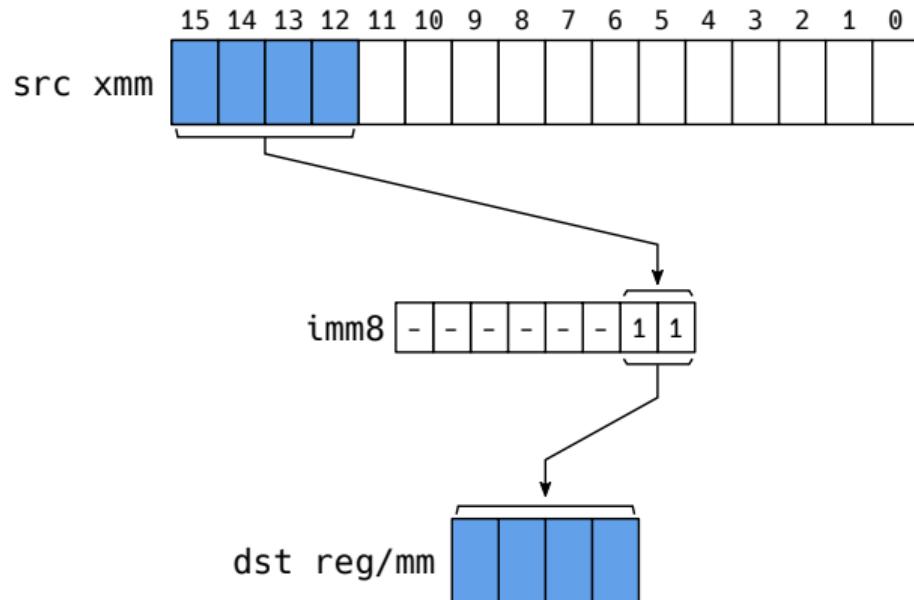
## Ejemplo - PEXTRB dst, src , imm8



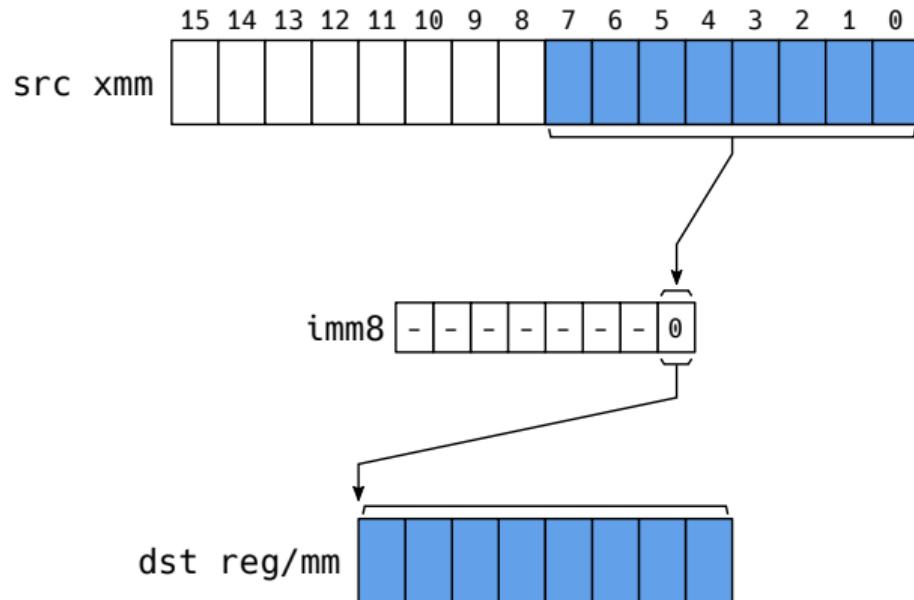
## Ejemplo - PEXTRW dst, src , imm8



## Ejemplo - PEXTRD dst, src , imm8



## Ejemplo - PEXTRQ dst, src , imm8



# Blend

Las instrucciones de *Blend* permiten **mezclar** registros dependiendo del valor de sus datos. Usando tanto inmediatos como otros registros.

# Blend

Las instrucciones de *Blend* permiten **mezclar** registros dependiendo del valor de sus datos. Usando tanto inmediatos como otros registros.

- **BLENDPS** - Blend Packed Single FP Values
- **BLENDPD** - Blend Packed Double FP Values
- **BLENDVPS** - Variable Blend Packed Single FP Values
- **BLENDVPD** - Variable Blend Packed Double FP Values
- **PBLENDW** - Blend Packed Words
- **PBLENDVB** - Variable Blend Packed Bytes

# Blend

## BLENDPS – Blend Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 OF 3A 0C /r ib BLENDPS <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Select packed single precision floating-point values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .

## BLENDPD – Blend Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 OF 3A 0D /r ib BLENDPD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Select packed DP-FP values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .

# Blend

## BLENDPS – Blend Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 OF 3A 0C /r ib BLENDPS <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Select packed single precision floating-point values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .

## BLENDPD – Blend Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 OF 3A 0D /r ib BLENDPD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Select packed DP-FP values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>imm8</i>

```

IF (IMM8[0] = 0) THEN DEST[31:0] ← DEST[31:0]
    ELSE DEST [31:0] ← SRC[31:0] FI
IF (IMM8[1] = 0) THEN DEST[63:32] ← DEST[63:32]
    ELSE DEST [63:32] ← SRC[63:32] FI
IF (IMM8[2] = 0) THEN DEST[95:64] ← DEST[95:64]
    ELSE DEST [95:64] ← SRC[95:64] FI
IF (IMM8[3] = 0) THEN DEST[127:96] ← DEST[127:96]
    ELSE DEST [127:96] ← SRC[127:96] FI
DEST[VLMAX-1:128] (Unmodified)

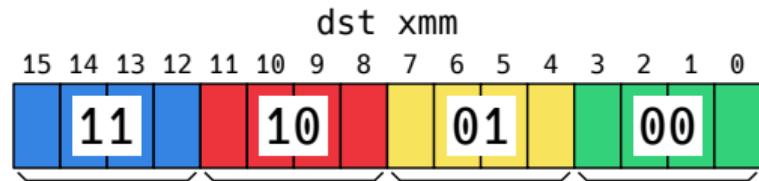
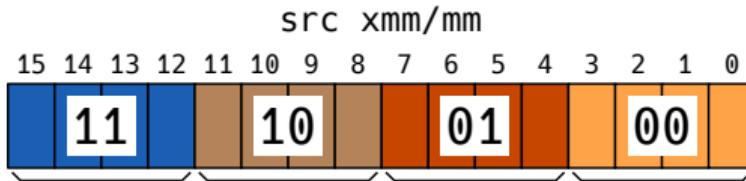
```

```

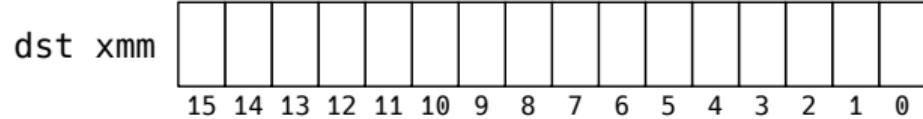
IF (IMM8[0] = 0) THEN DEST[63:0] ← DEST[63:0]
    ELSE DEST [63:0] ← SRC[63:0] FI
IF (IMM8[1] = 0) THEN DEST[127:64] ← DEST[127:64]
    ELSE DEST [127:64] ← SRC[127:64] FI
DEST[VLMAX-1:128] (Unmodified)

```

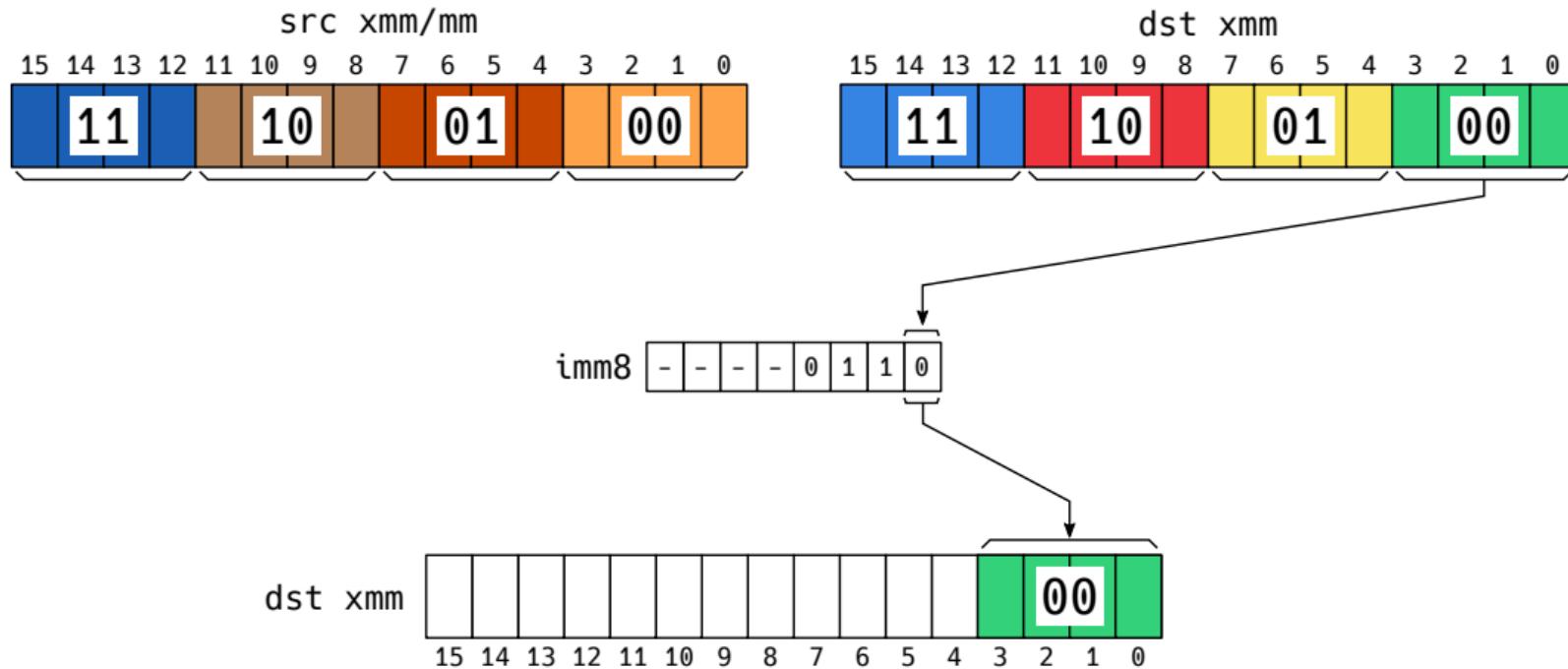
## Ejemplo - BLENDPS dst, src , imm8



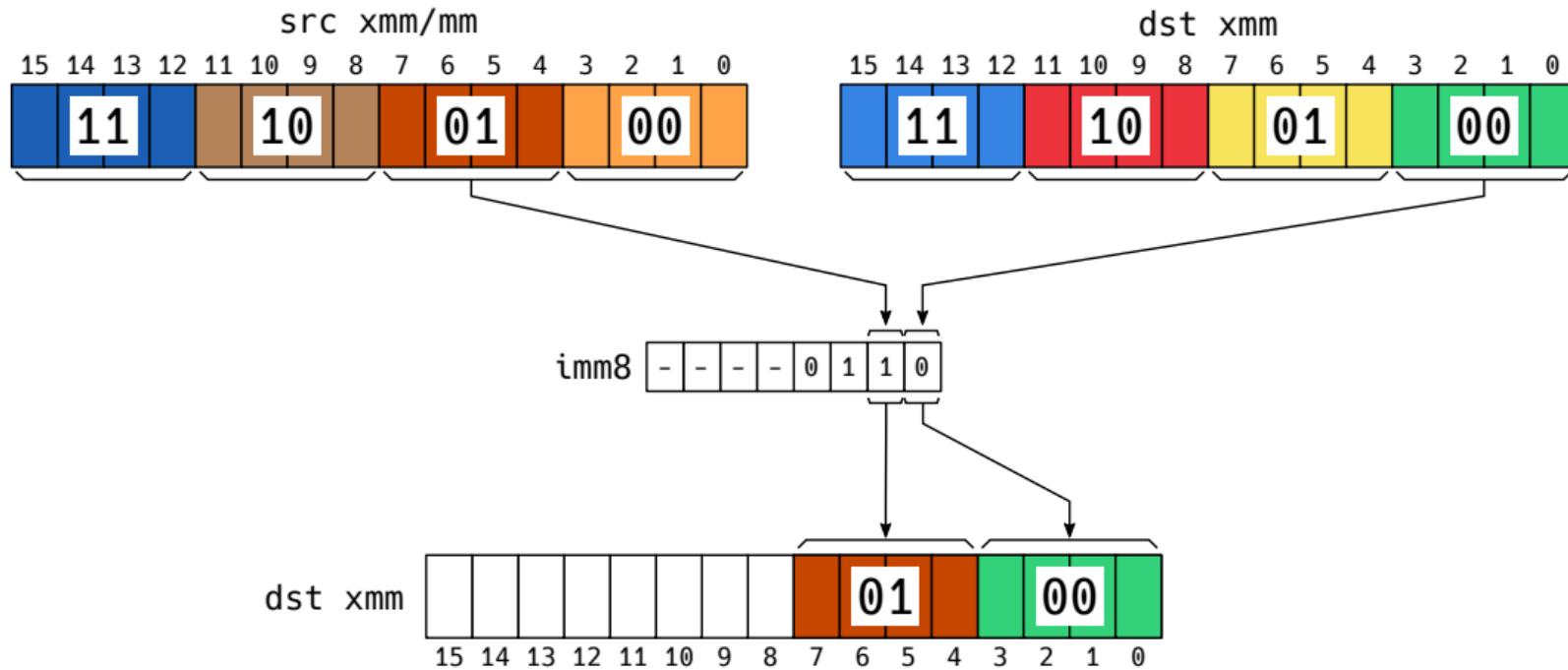
imm8 [- - - - 0 1 1 0]



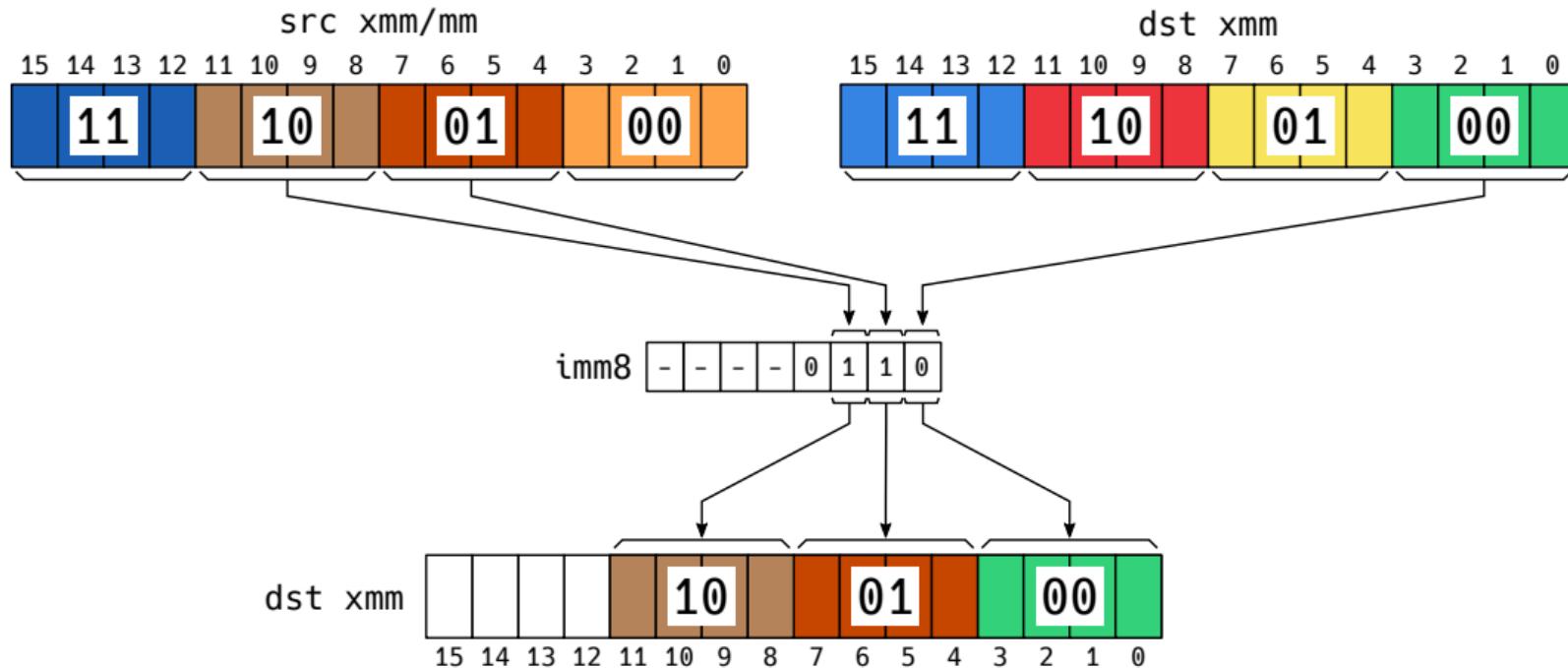
## Ejemplo - BLENDPS dst, src , imm8



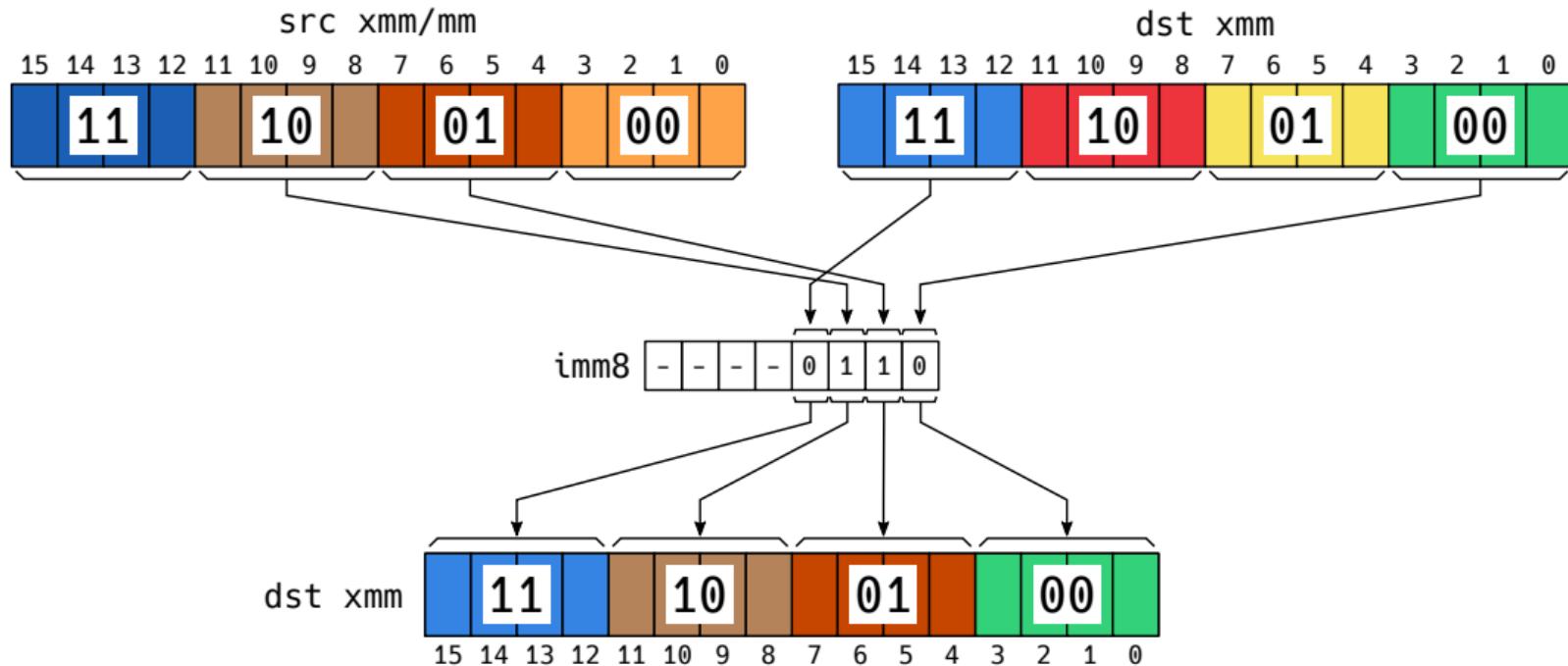
## Ejemplo - BLENDPS dst, src , imm8



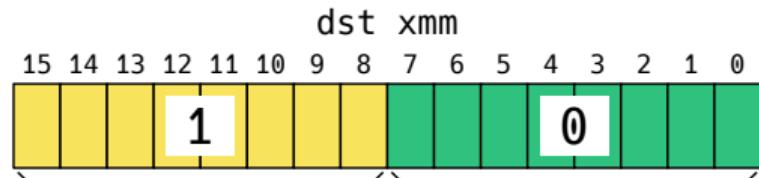
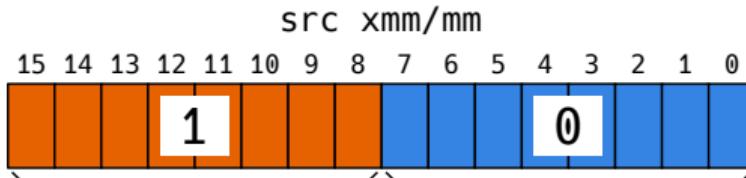
## Ejemplo - BLENDPS dst, src , imm8



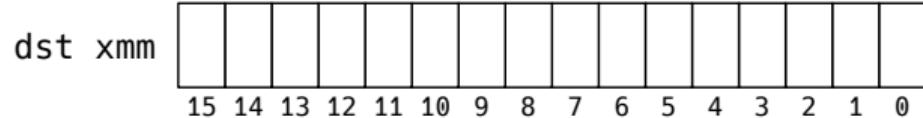
## Ejemplo - BLENDPS dst, src , imm8



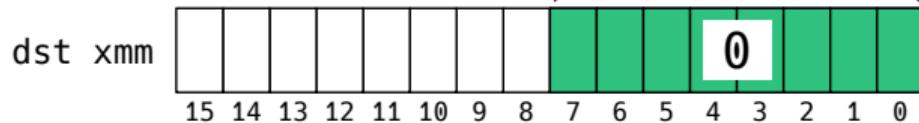
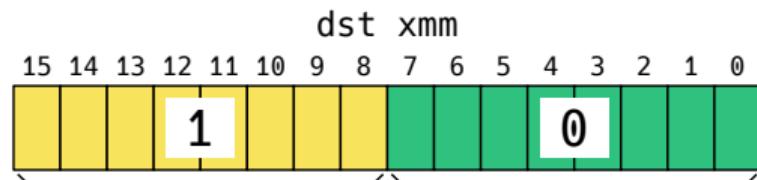
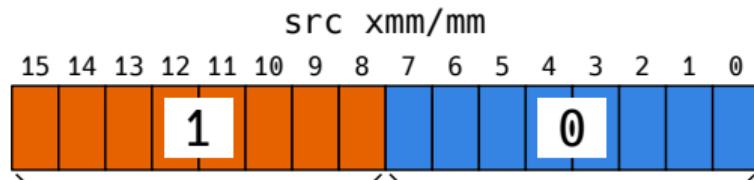
## Ejemplo - BLENDPD dst, src , imm8



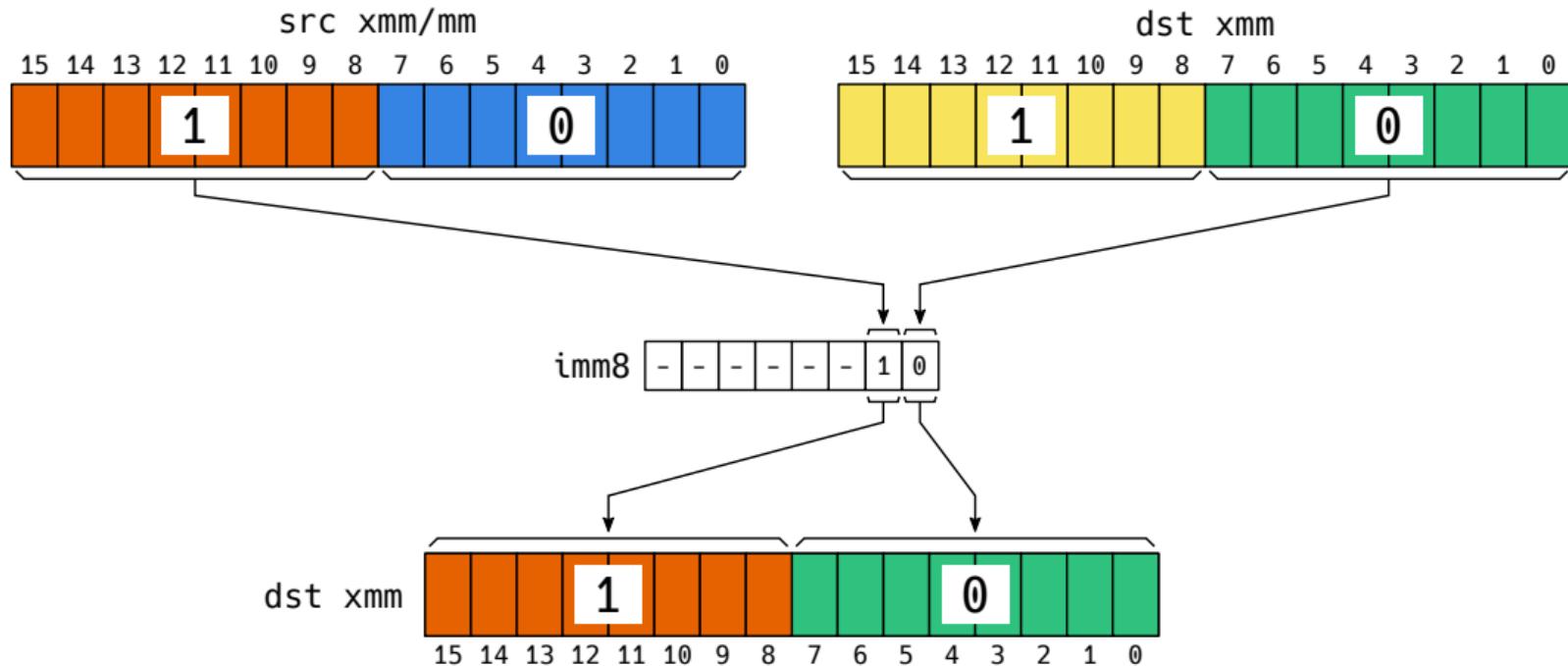
imm8 [- - - - - - 1 0]



## Ejemplo - BLENDPD dst, src , imm8



## Ejemplo - BLENDPD dst, src , imm8



# Blend

## BLENDVPS – Variable Blend Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 14 /r  BLENDVPS <i>xmm1, xmm2/m128, &lt;XMM0&gt;</i>	RMO	V/V	SSE4_1	Select packed single precision floating-point values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>XMM0</i> and store the values into <i>xmm1</i> .

## BLENDVPD – Variable Blend Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 15 /r  BLENDVPD <i>xmm1, xmm2/m128, &lt;XMM0&gt;</i>	RMO	V/V	SSE4_1	Select packed DP FP values from <i>xmm1</i> and <i>xmm2</i> from mask specified in <i>XMM0</i> and store the values in <i>xmm1</i> .

# Blend

## BLENDVPS – Variable Blend Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 14 /r  BLENDVPS $xmm1, xmm2/m128, <XMM0>$	RMO	V/V	SSE4_1	Select packed single precision floating-point values from $xmm1$ and $xmm2/m128$ from mask specified in $XMM0$ and store the values into $xmm1$ .

## BLENDVPD – Variable Blend Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 15 /r  BLENDVPD $xmm1, xmm2/m128, <XMM0>$  BLENDVPS	RMO	V/V	SSE4_1	Select packed DP FP values from $xmm1$ and $xmm2$ from mask specified in $XMM0$ and store the values in $xmm1$ .

### BLENDVPS (128-bit Legacy SSE version)

```

MASK ← XMM0
IF (MASK[31] = 0) THEN DEST[31:0] ← DEST[31:0]
    ELSE DEST [31:0] ← SRC[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] ← DEST[63:32]
    ELSE DEST [63:32] ← SRC[63:32] FI
IF (MASK[95] = 0) THEN DEST[95:64] ← DEST[95:64]
    ELSE DEST [95:64] ← SRC[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] ← DEST[127:96]
    ELSE DEST [127:96] ← SRC[127:96] FI
DEST[VLMAX-1:128] (Unmodified)

```

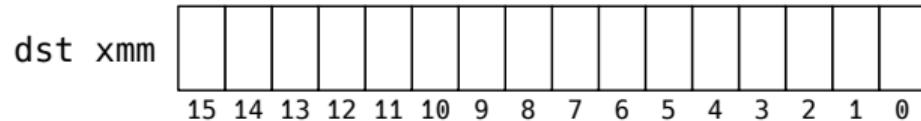
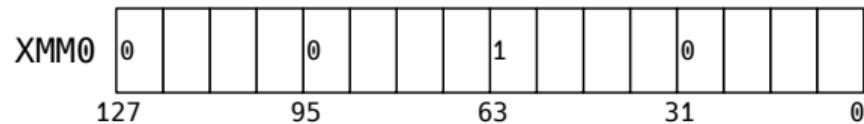
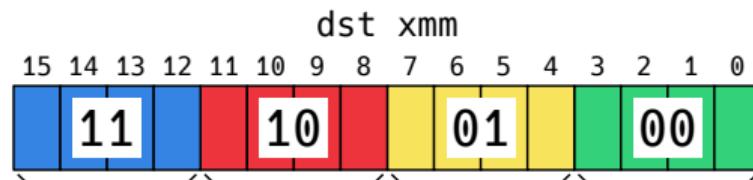
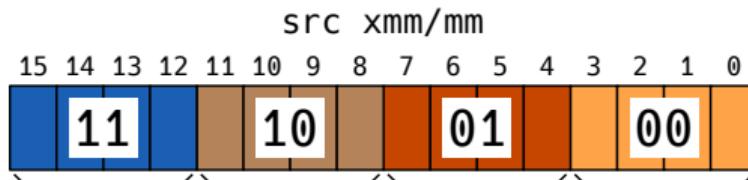
### BLENDVPD (128-bit Legacy SSE version)

```

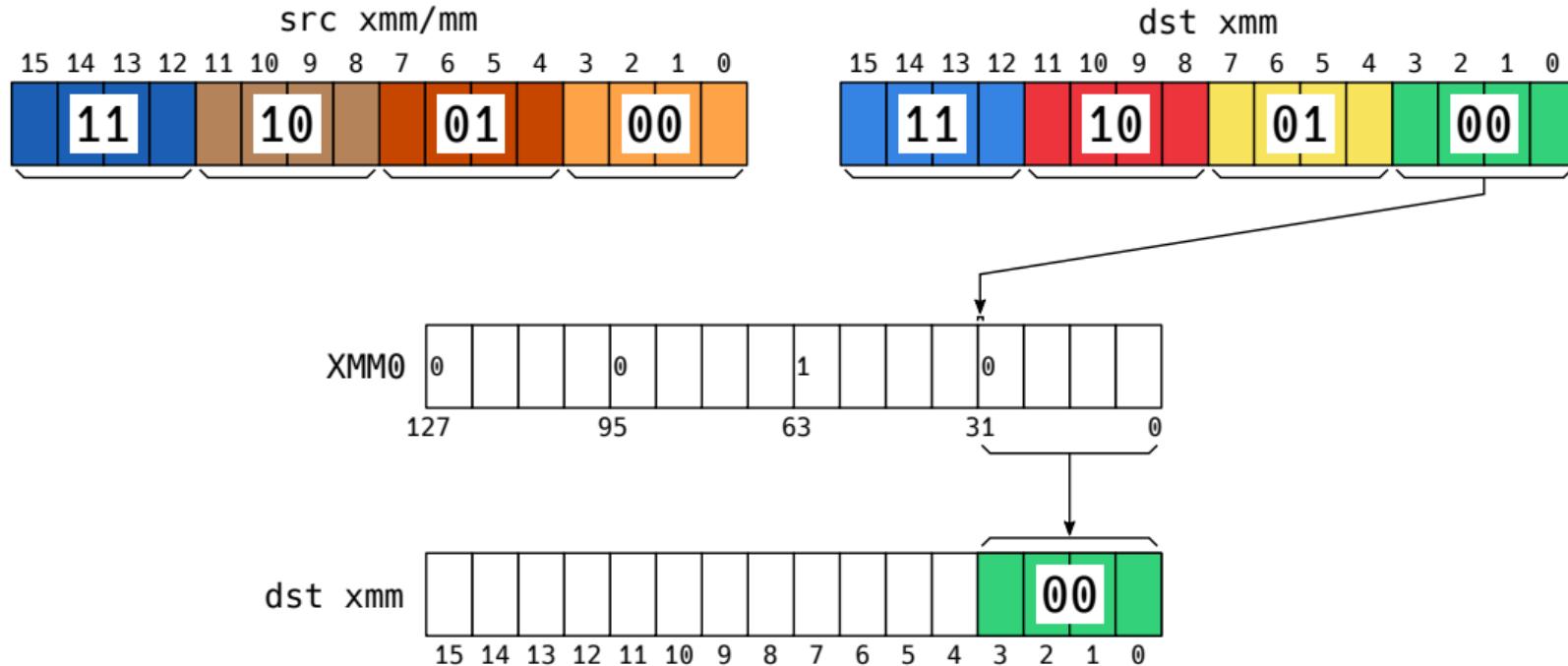
MASK ← XMM0
IF (MASK[63] = 0) THEN DEST[63:0] ← DEST[63:0]
    ELSE DEST [63:0] ← SRC[63:0] FI
IF (MASK[127] = 0) THEN DEST[127:64] ← DEST[127:64]
    ELSE DEST [127:64] ← SRC[127:64] FI
DEST[VLMAX-1:128] (Unmodified)

```

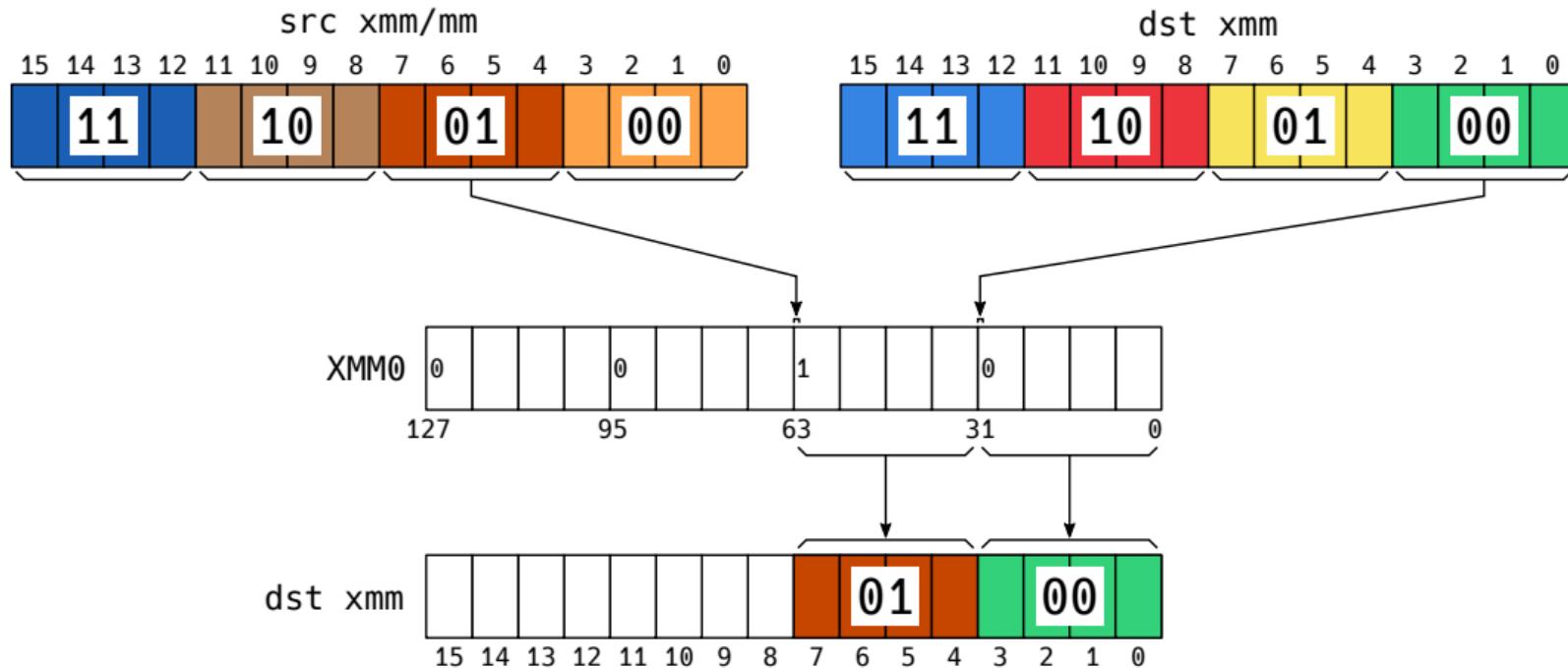
### Ejemplo - BLENDVPS dst, src , imm8



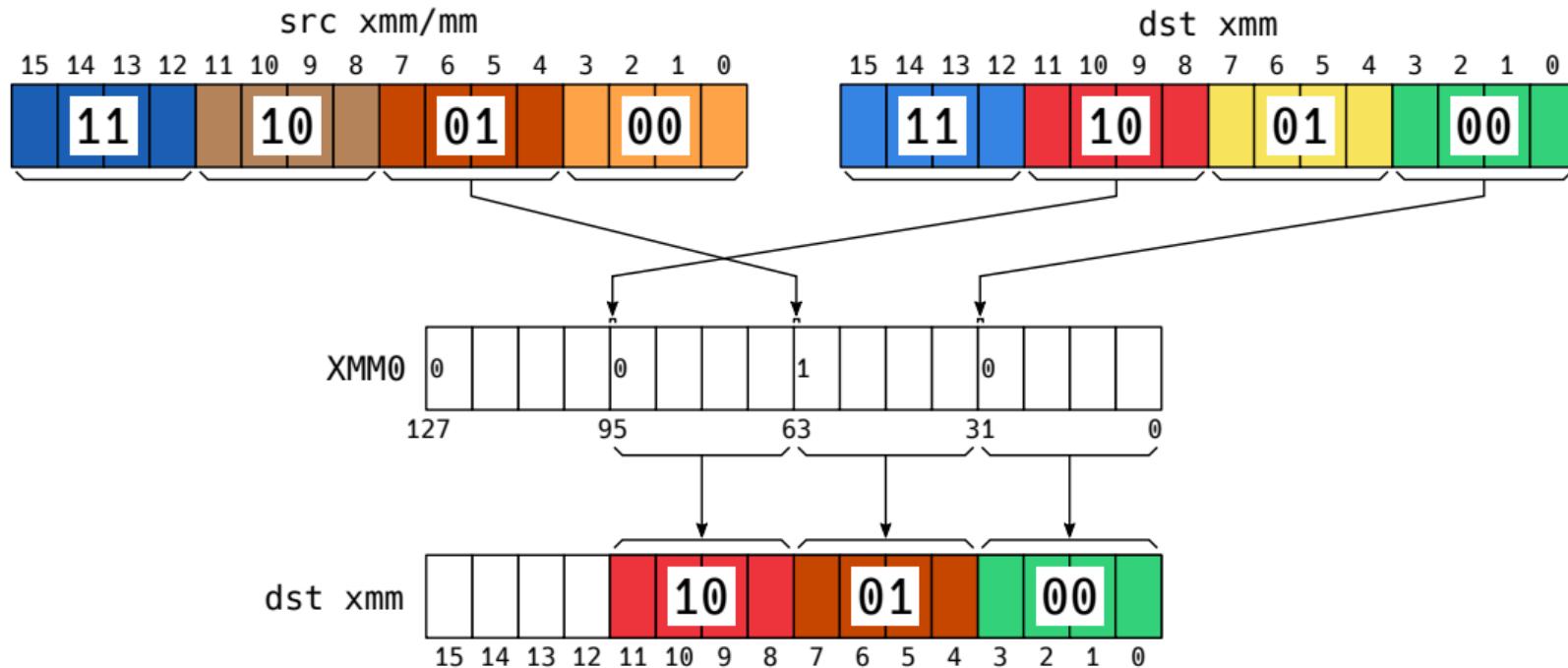
# Ejemplo - BLENDVPS dst, src , imm8



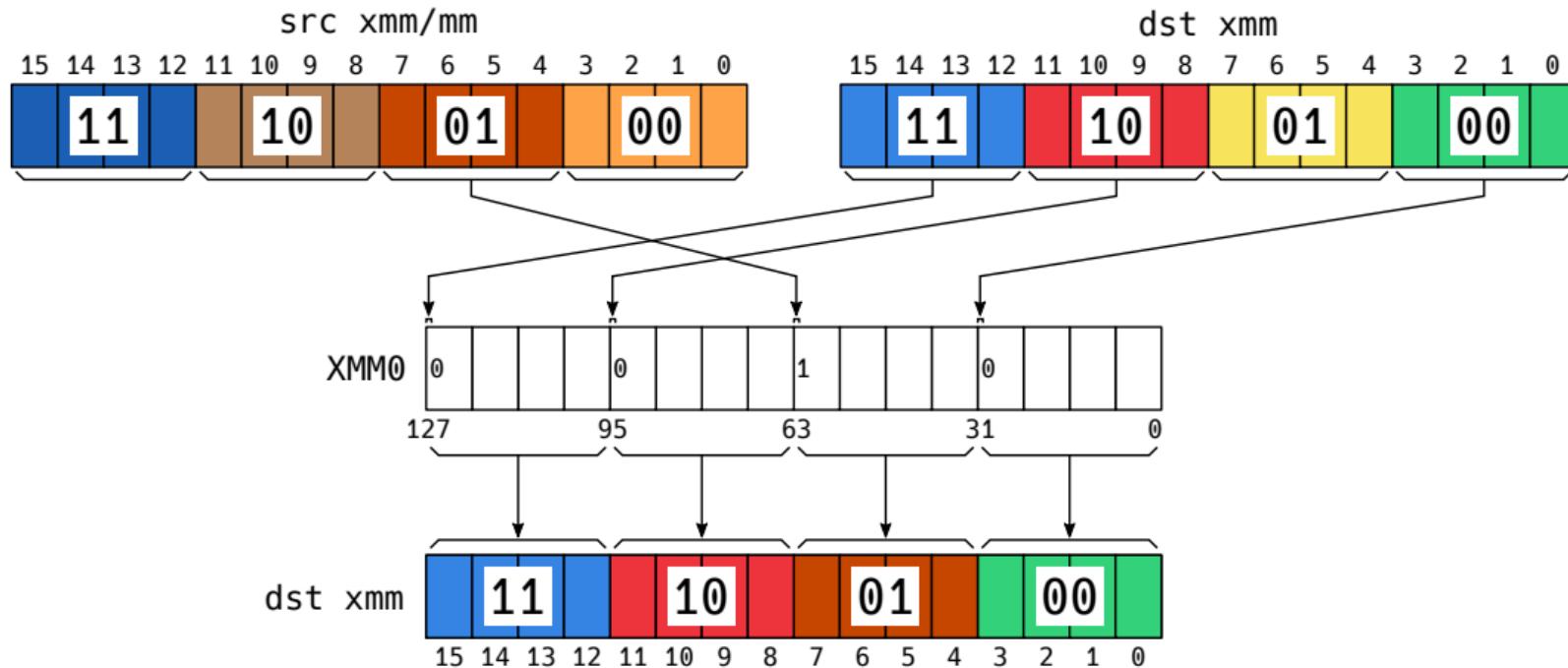
# Ejemplo - BLENDVPS dst, src , imm8



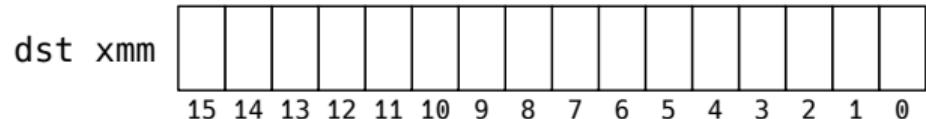
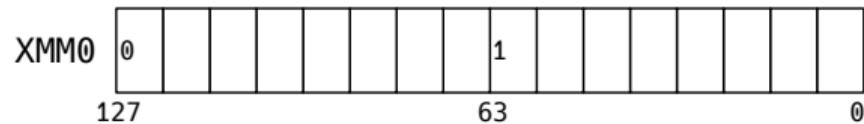
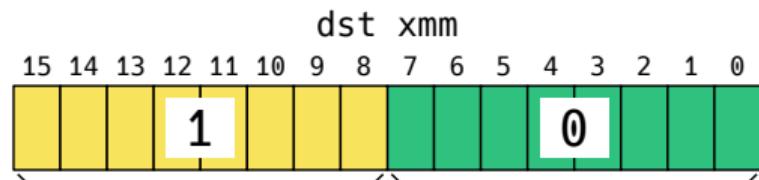
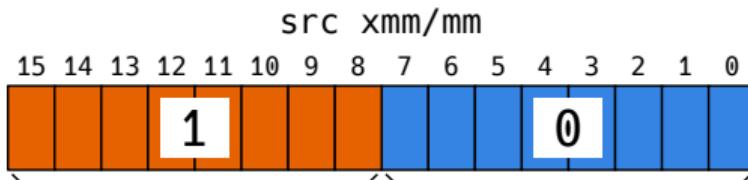
# Ejemplo - BLENDVPS dst, src , imm8



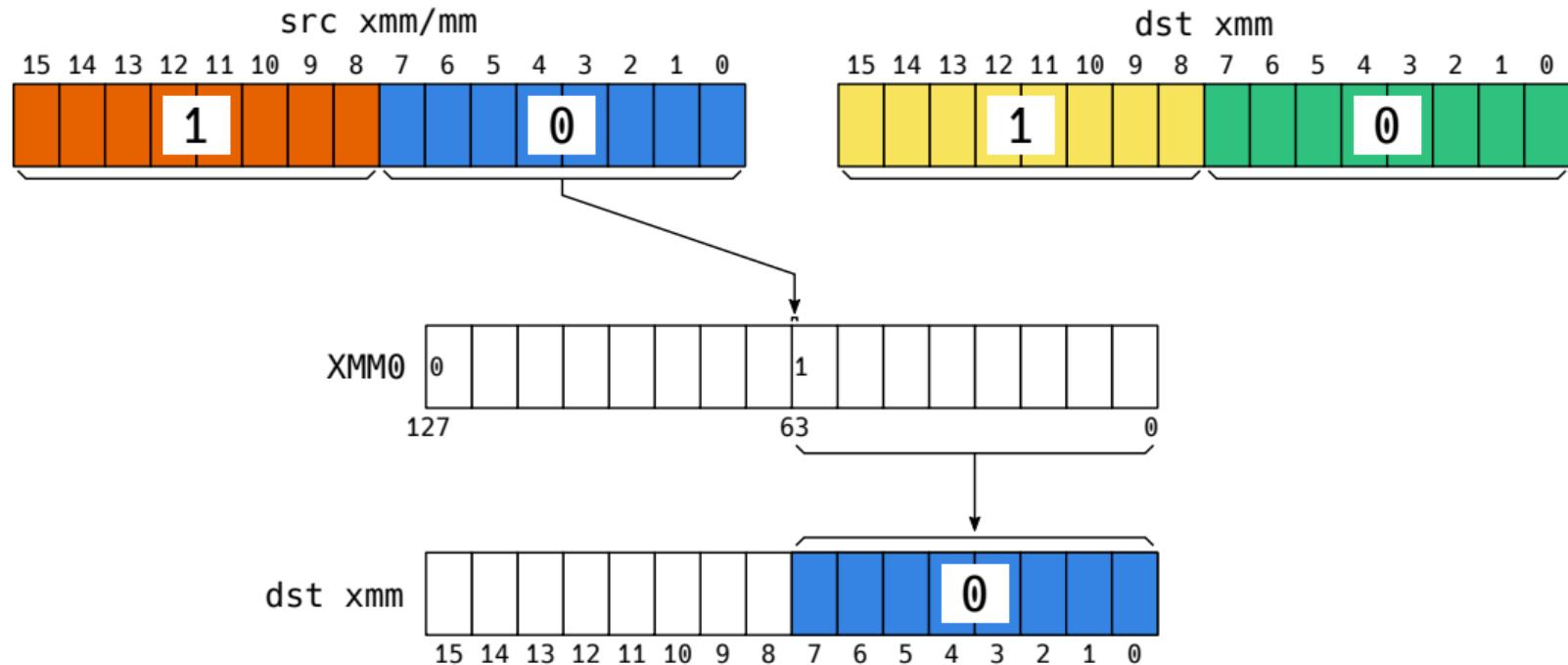
# Ejemplo - BLENDVPS dst, src , imm8



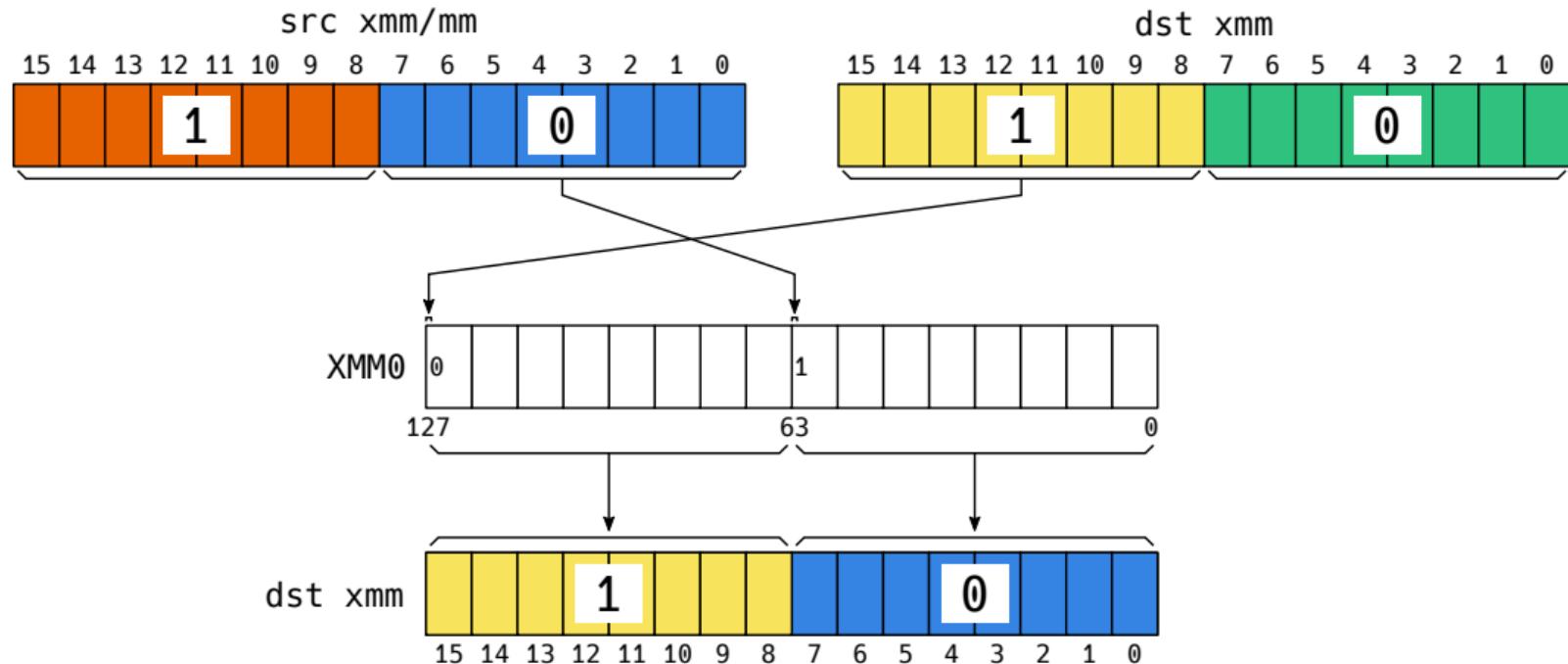
## Ejemplo - BLENDVPD dst, src , imm8



## Ejemplo - BLENDVPD dst, src , imm8



## Ejemplo - BLENDVPD dst, src , imm8



## Blend

## PBLENDW – Blend Packed Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0E /r ib PBLENDW <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Select words from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .

# Blend

## PBLENDW – Blend Packed Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0E /r ib PBLENDW <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Select words from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .

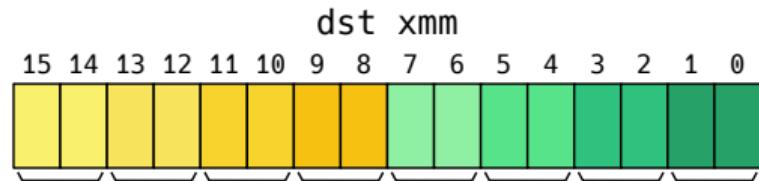
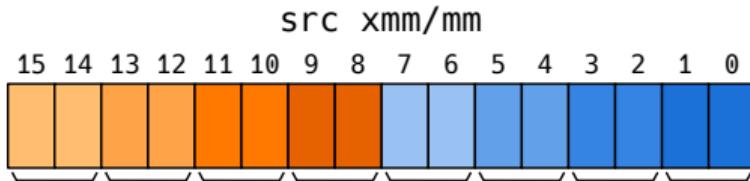
### Operation

#### PBLENDW (128-bit Legacy SSE version)

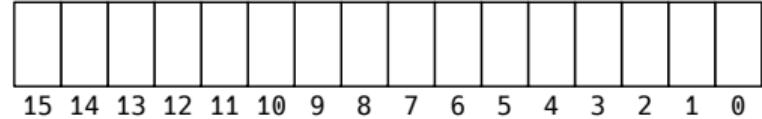
```

IF (imm8[0] = 1) THEN DEST[15:0] ← SRC[15:0]
ELSE DEST[15:0] ← DEST[15:0]
IF (imm8[1] = 1) THEN DEST[31:16] ← SRC[31:16]
ELSE DEST[31:16] ← DEST[31:16]
IF (imm8[2] = 1) THEN DEST[47:32] ← SRC[47:32]
ELSE DEST[47:32] ← DEST[47:32]
IF (imm8[3] = 1) THEN DEST[63:48] ← SRC[63:48]
ELSE DEST[63:48] ← DEST[63:48]
IF (imm8[4] = 1) THEN DEST[79:64] ← SRC[79:64]
ELSE DEST[79:64] ← DEST[79:64]
IF (imm8[5] = 1) THEN DEST[95:80] ← SRC[95:80]
ELSE DEST[95:80] ← DEST[95:80]
IF (imm8[6] = 1) THEN DEST[111:96] ← SRC[111:96]
ELSE DEST[111:96] ← DEST[111:96]
IF (imm8[7] = 1) THEN DEST[127:112] ← SRC[127:112]
```

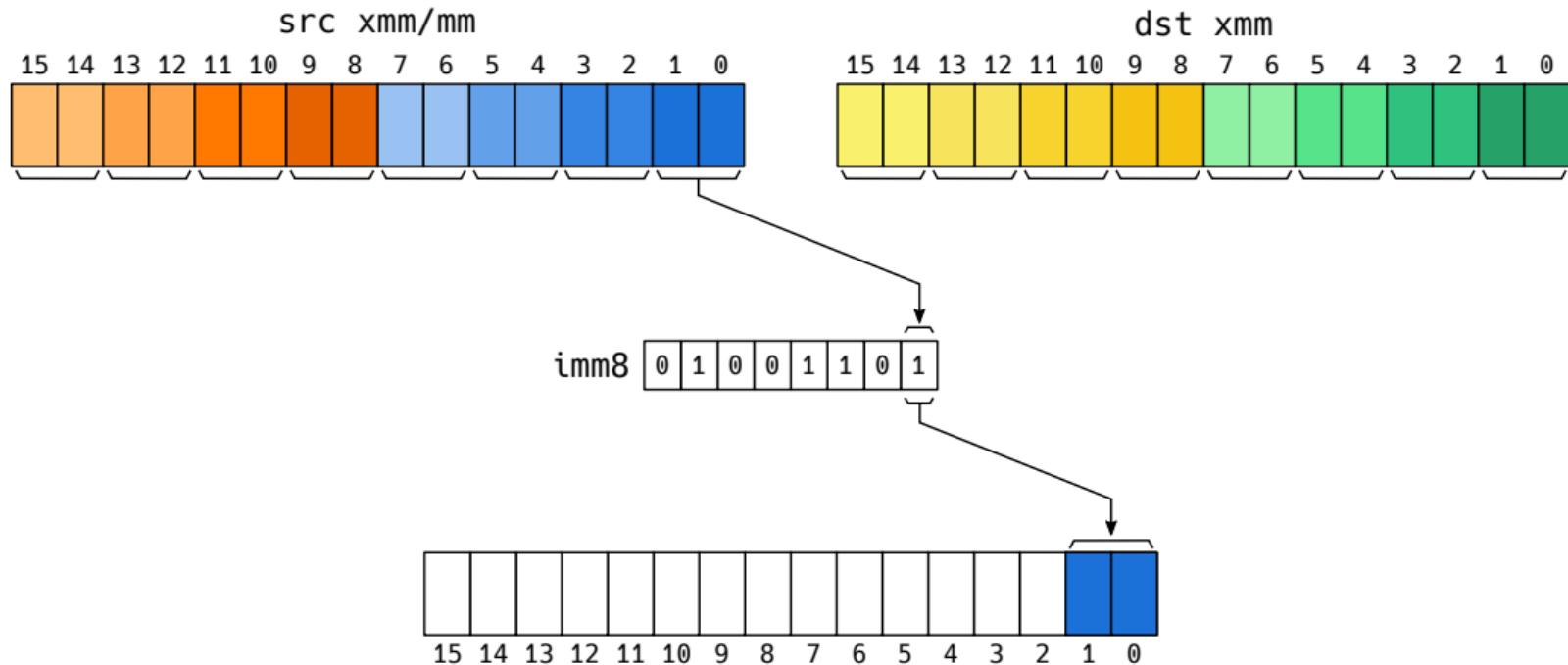
## Ejemplo - PBLENDW dst, src , imm8



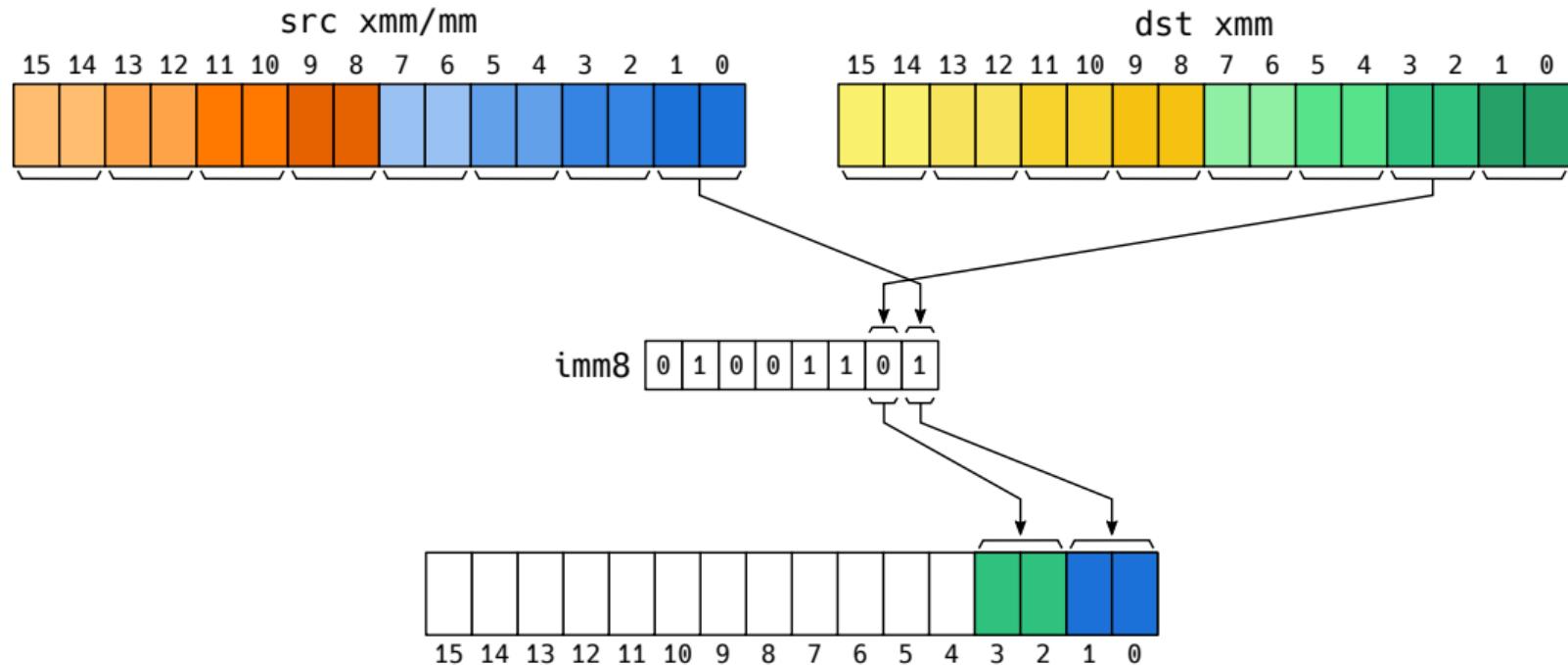
imm8 [0 | 1 | 0 | 0 | 1 | 1 | 0 | 1]



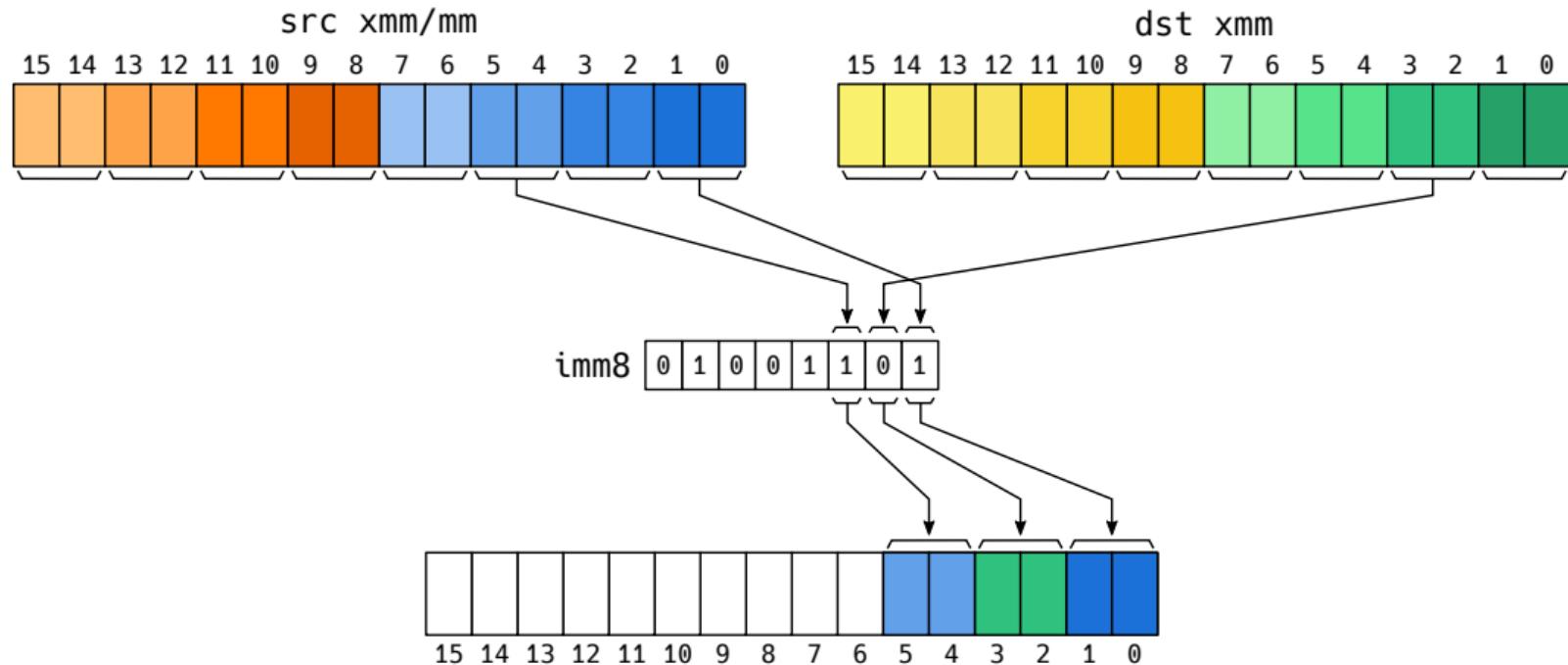
## Ejemplo - PBLENDW dst, src , imm8



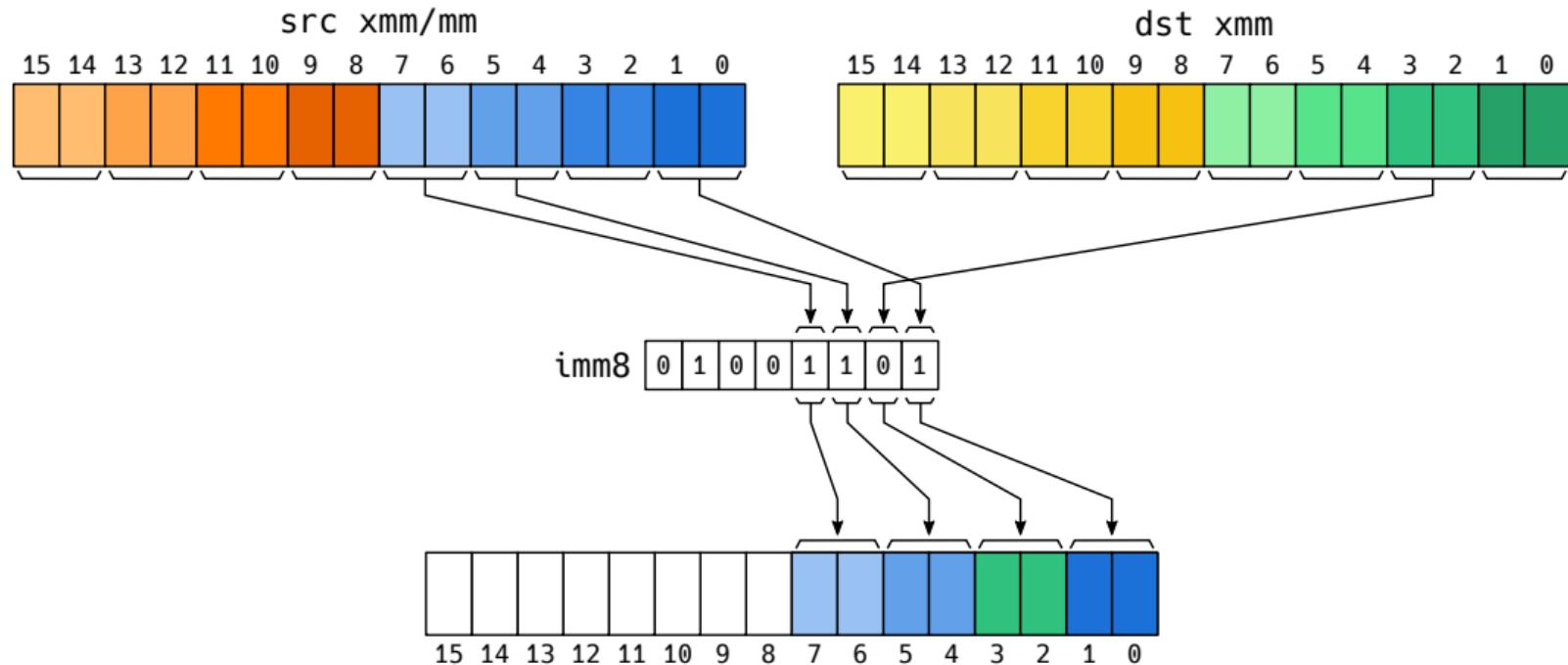
## Ejemplo - PBLENDW dst, src , imm8



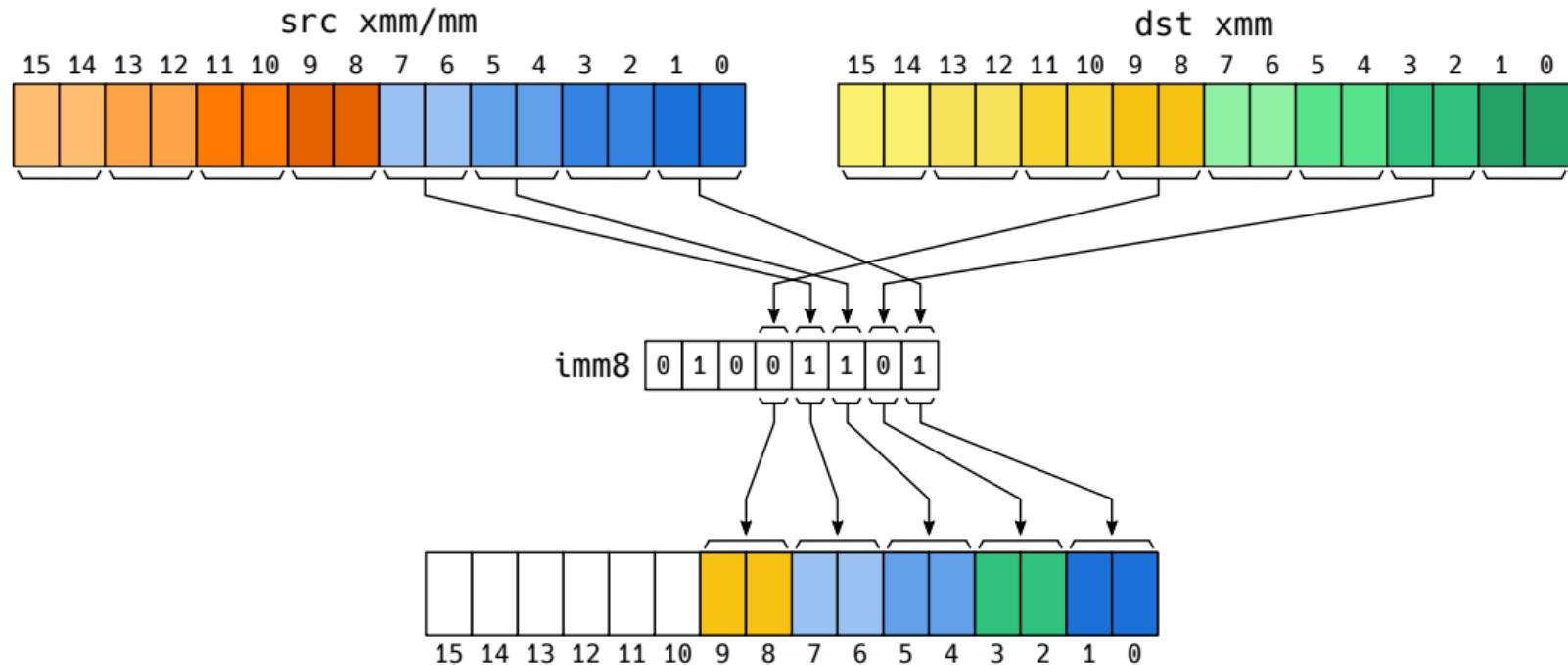
## Ejemplo - PBLENDW dst, src , imm8



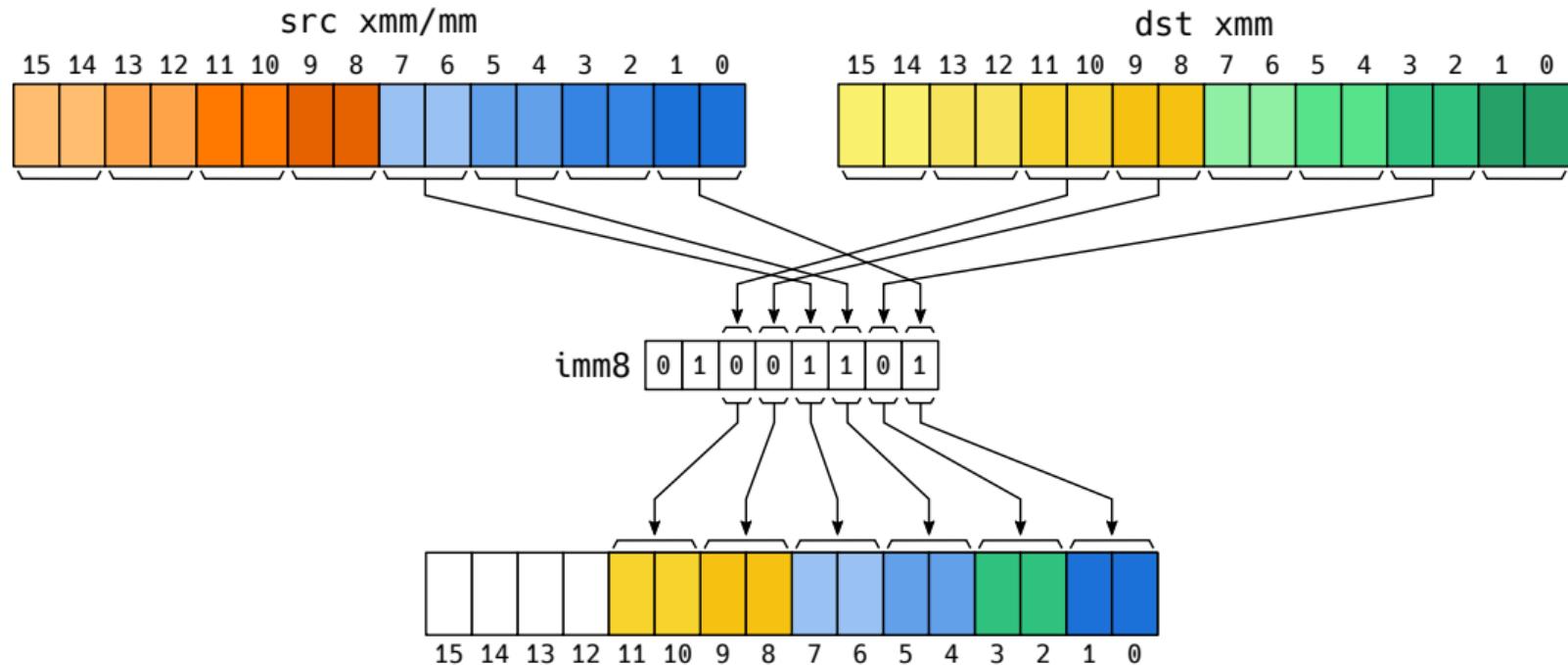
## Ejemplo - PBLENDW dst, src , imm8



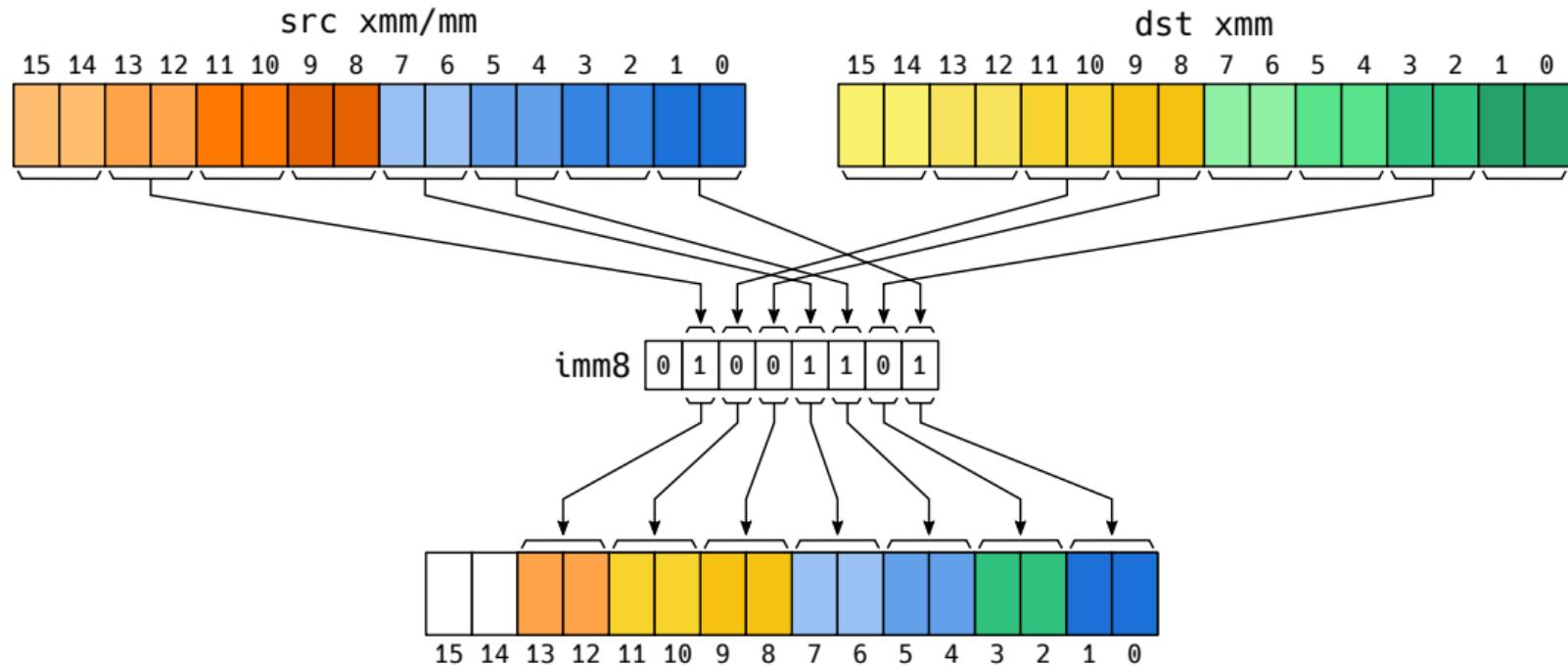
## Ejemplo - PBLENDW dst, src , imm8



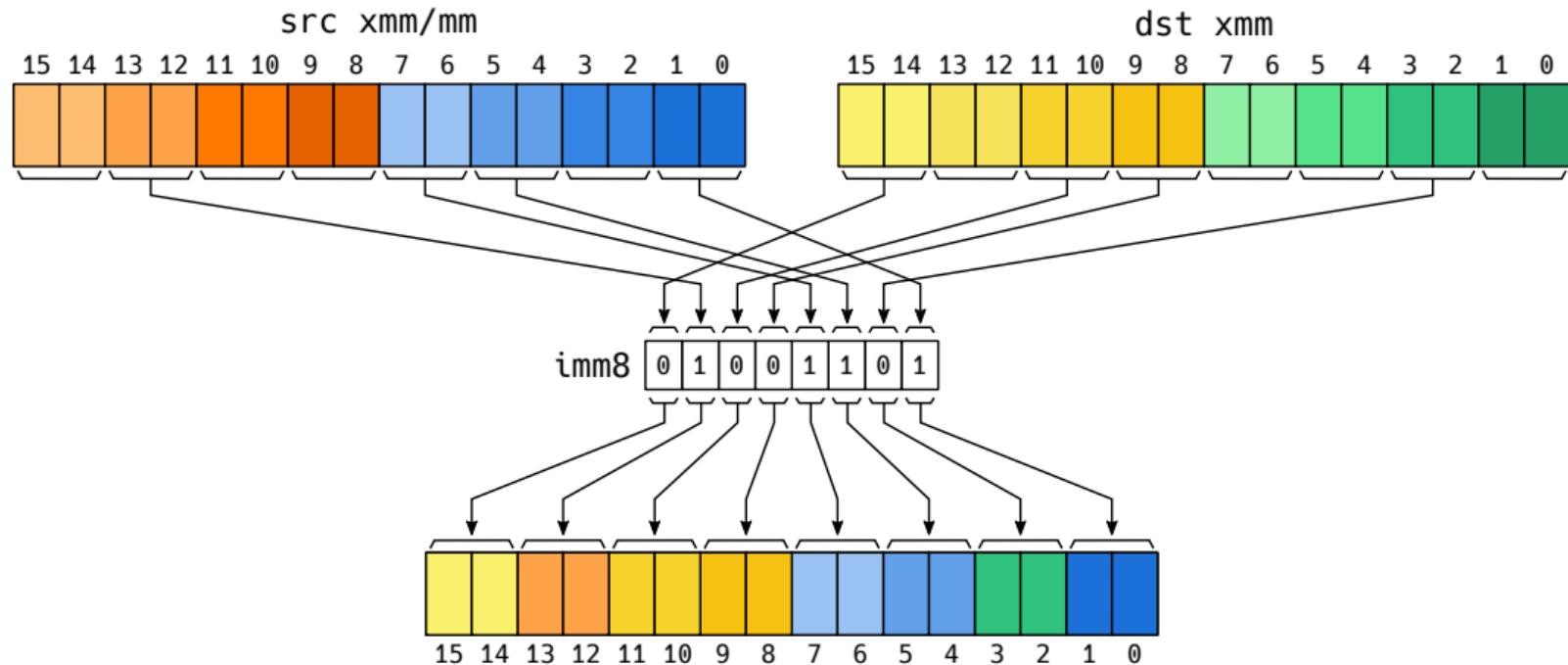
## Ejemplo - PBLENDW dst, src , imm8



## Ejemplo - PBLENDW dst, src , imm8



## Ejemplo - PBLENDW dst, src , imm8



# Blend

## PBLENDVB – Variable Blend Packed Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 10 /r PBLENDVB <i>xmm1, xmm2/m128, &lt;XMM0&gt;</i>	RM	V/V	SSE4_1	Select byte values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in the high bit of each byte in <i>XMM0</i> and store the values into <i>xmm1</i> .

## Blend

## PBLENDVB – Variable Blend Packed Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 10 /r PBLENDVB <i>xmm1, xmm2/m128, &lt;XMM0&gt;</i>	RM	V/V	SSE4_1	Select byte values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in the high bit of each byte in <i>XMM0</i> and store the values into <i>xmm1</i> .

## Operation

## PBLENDVB (128-bit Legacy SSE version)

```

MASK ← XMM0
IF (MASK[7] = 1) THEN DEST[7:0] ← SRC[7:0];
ELSE DEST[7:0] ← DEST[7:0];
IF (MASK[15] = 1) THEN DEST[15:8] ← SRC[15:8];
ELSE DEST[15:8] ← DEST[15:8];
IF (MASK[23] = 1) THEN DEST[23:16] ← SRC[23:16]
ELSE DEST[23:16] ← DEST[23:16];
IF (MASK[31] = 1) THEN DEST[31:24] ← SRC[31:24]
ELSE DEST[31:24] ← DEST[31:24];
IF (MASK[39] = 1) THEN DEST[39:32] ← SRC[39:32]
ELSE DEST[39:32] ← DEST[39:32];
IF (MASK[47] = 1) THEN DEST[47:40] ← SRC[47:40]
ELSE DEST[47:40] ← DEST[47:40];
IF (MASK[55] = 1) THEN DEST[55:48] ← SRC[55:48]
ELSE DEST[55:48] ← DEST[55:48];
IF (MASK[63] = 1) THEN DEST[63:56] ← SRC[63:56]
ELSE DEST[63:56] ← DEST[63:56];

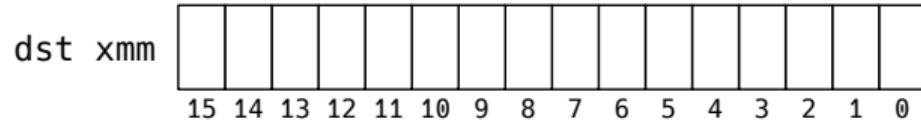
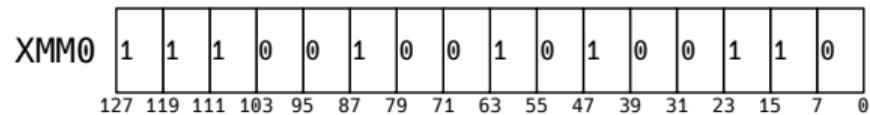
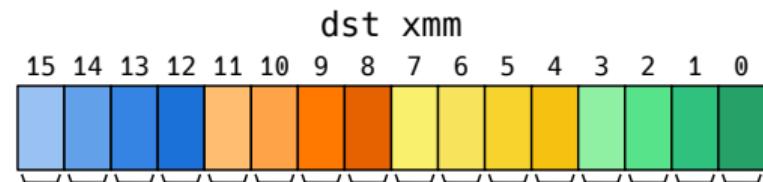
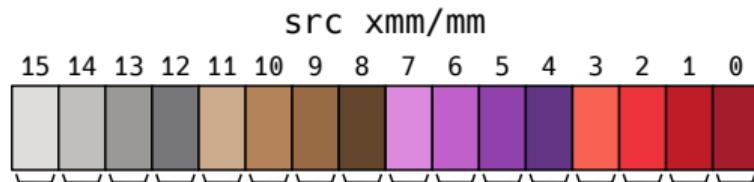
```

```

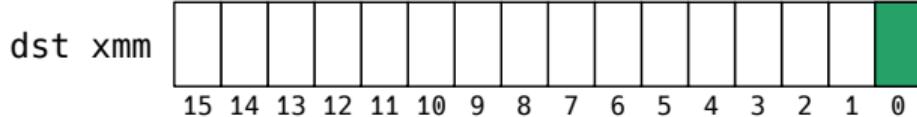
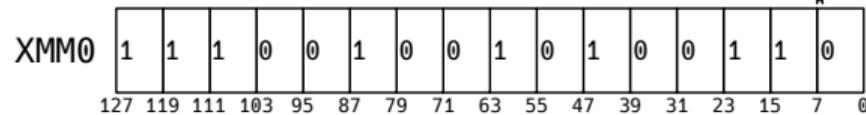
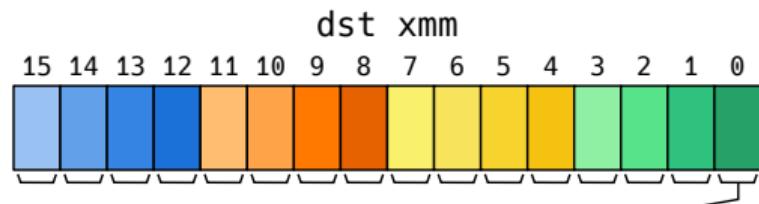
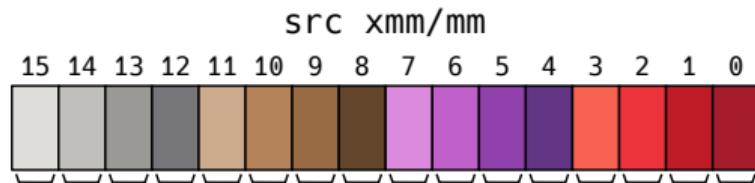
IF (MASK[71] = 1) THEN DEST[71:64] ← SRC[71:64]
ELSE DEST[71:64] ← DEST[71:64];
IF (MASK[79] = 1) THEN DEST[79:72] ← SRC[79:72]
ELSE DEST[79:72] ← DEST[79:72];
IF (MASK[87] = 1) THEN DEST[87:80] ← SRC[87:80]
ELSE DEST[87:80] ← DEST[87:80];
IF (MASK[95] = 1) THEN DEST[95:88] ← SRC[95:88]
ELSE DEST[95:88] ← DEST[95:88];
IF (MASK[103] = 1) THEN DEST[103:96] ← SRC[103:96]
ELSE DEST[103:96] ← DEST[103:96];
IF (MASK[111] = 1) THEN DEST[111:104] ← SRC[111:104]
ELSE DEST[111:104] ← DEST[111:104];
IF (MASK[119] = 1) THEN DEST[119:112] ← SRC[119:112]
ELSE DEST[119:112] ← DEST[119:112];
IF (MASK[127] = 1) THEN DEST[127:120] ← SRC[127:120]
ELSE DEST[127:120] ← DEST[127:120]);
DEST[VLMAX-1:128] (Unmodified)

```

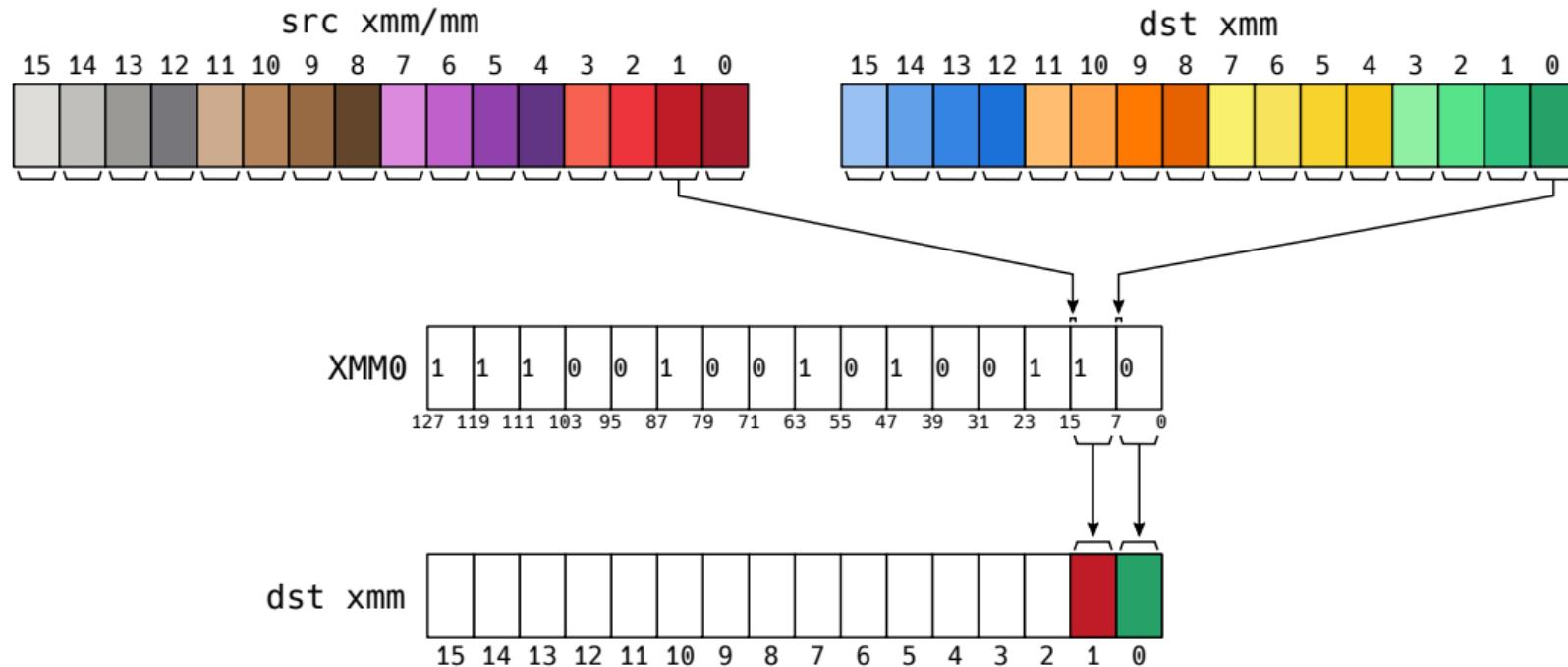
## Ejemplo - PBLENDVB dst, src



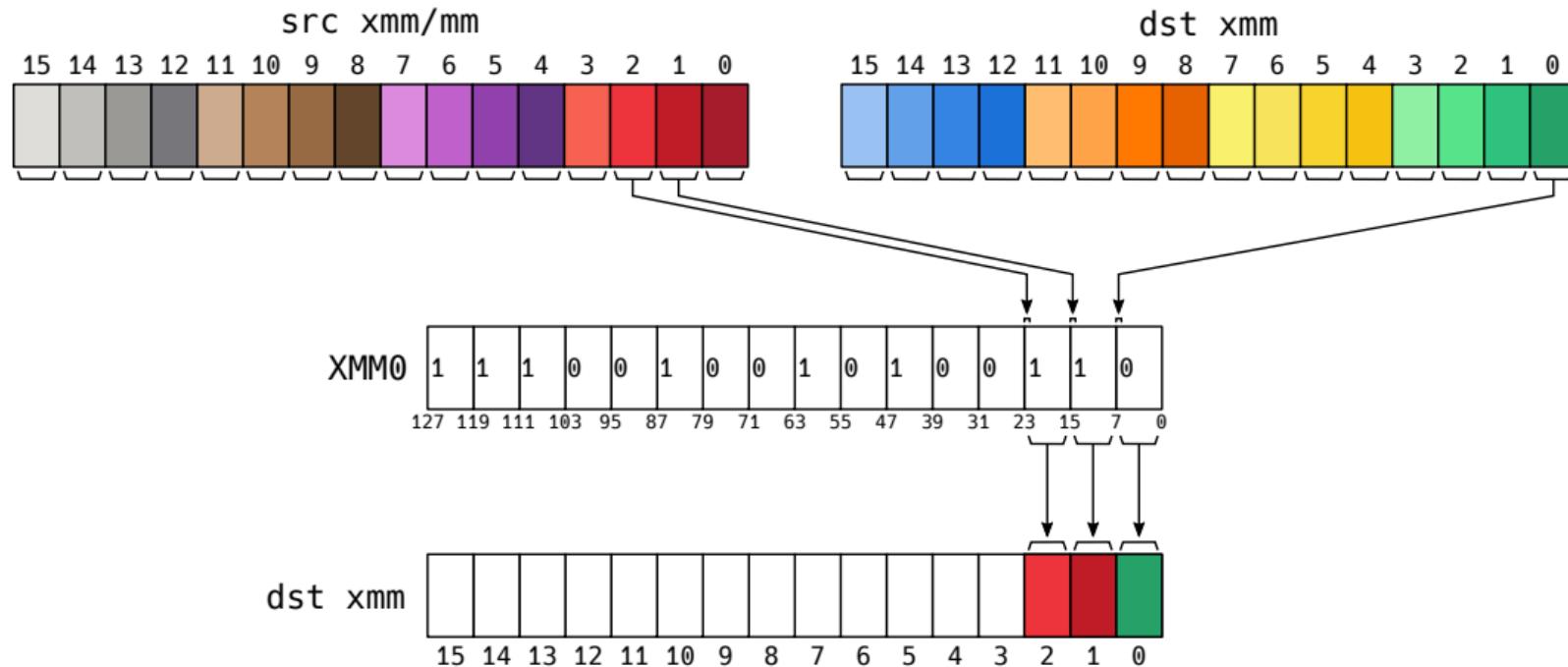
## Ejemplo - PBLENDVB dst, src



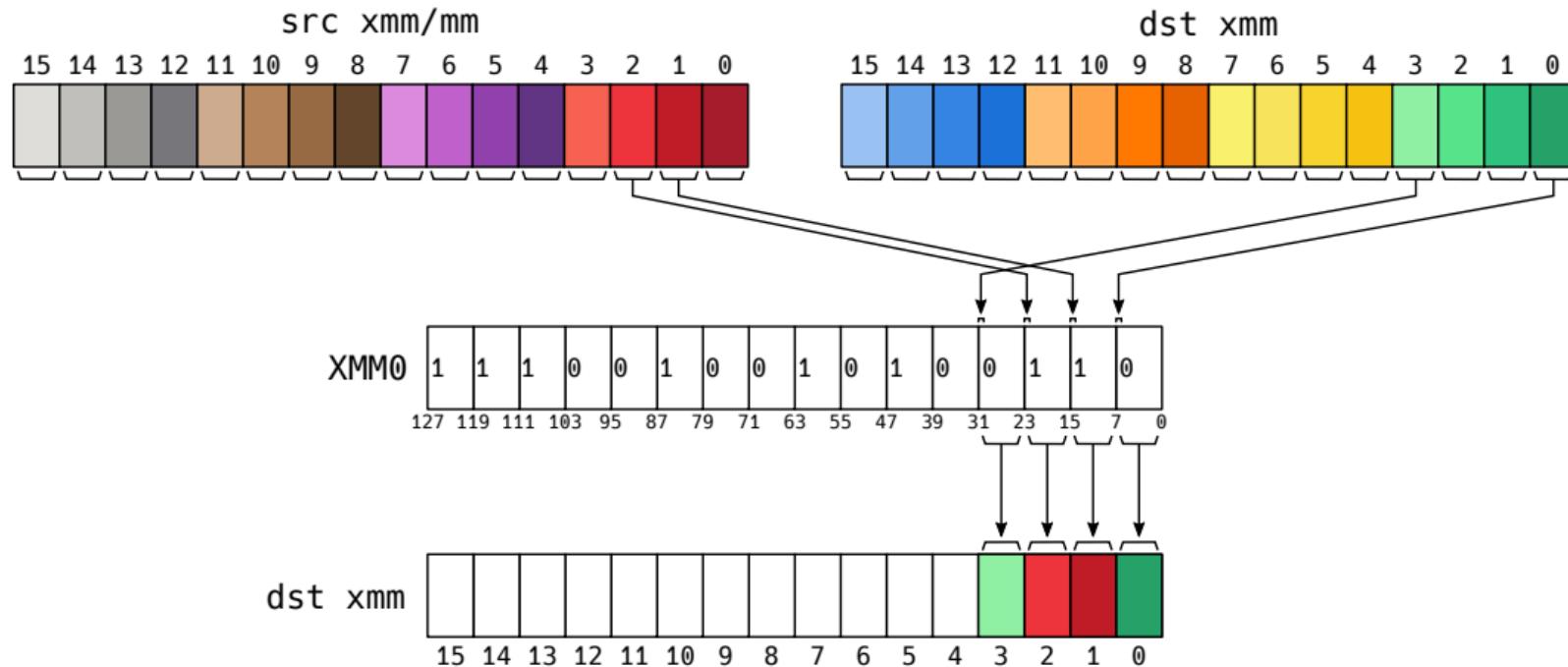
## Ejemplo - PBLENDVB dst, src



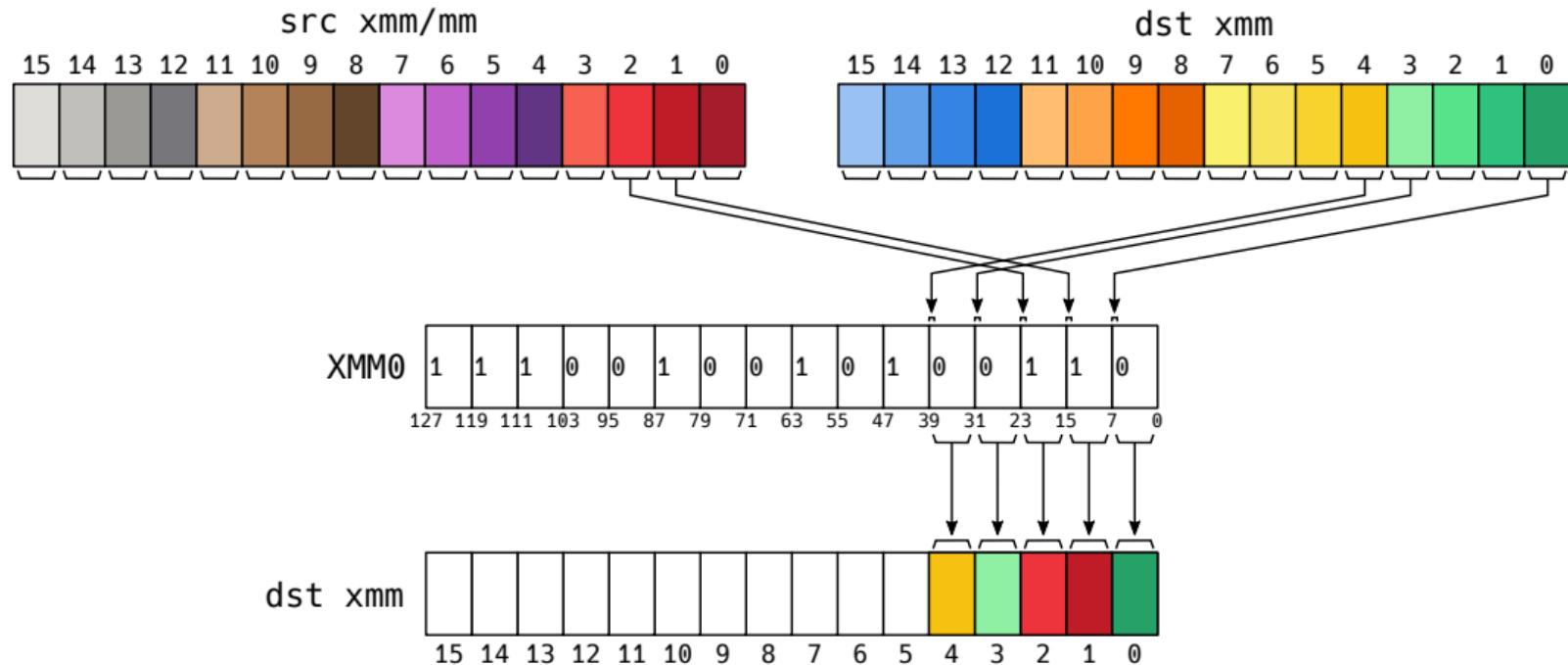
# Ejemplo - PBLENDVB dst, src



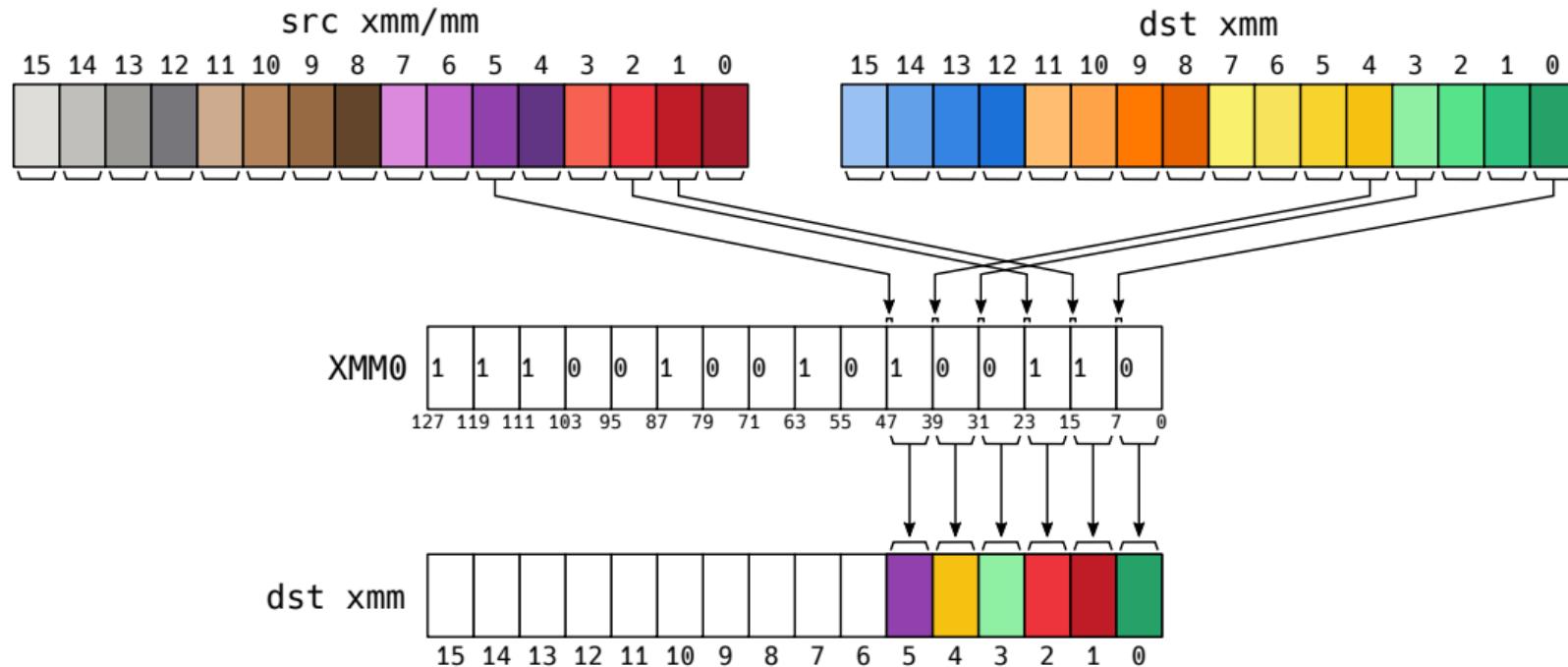
# Ejemplo - PBLENDVB dst, src



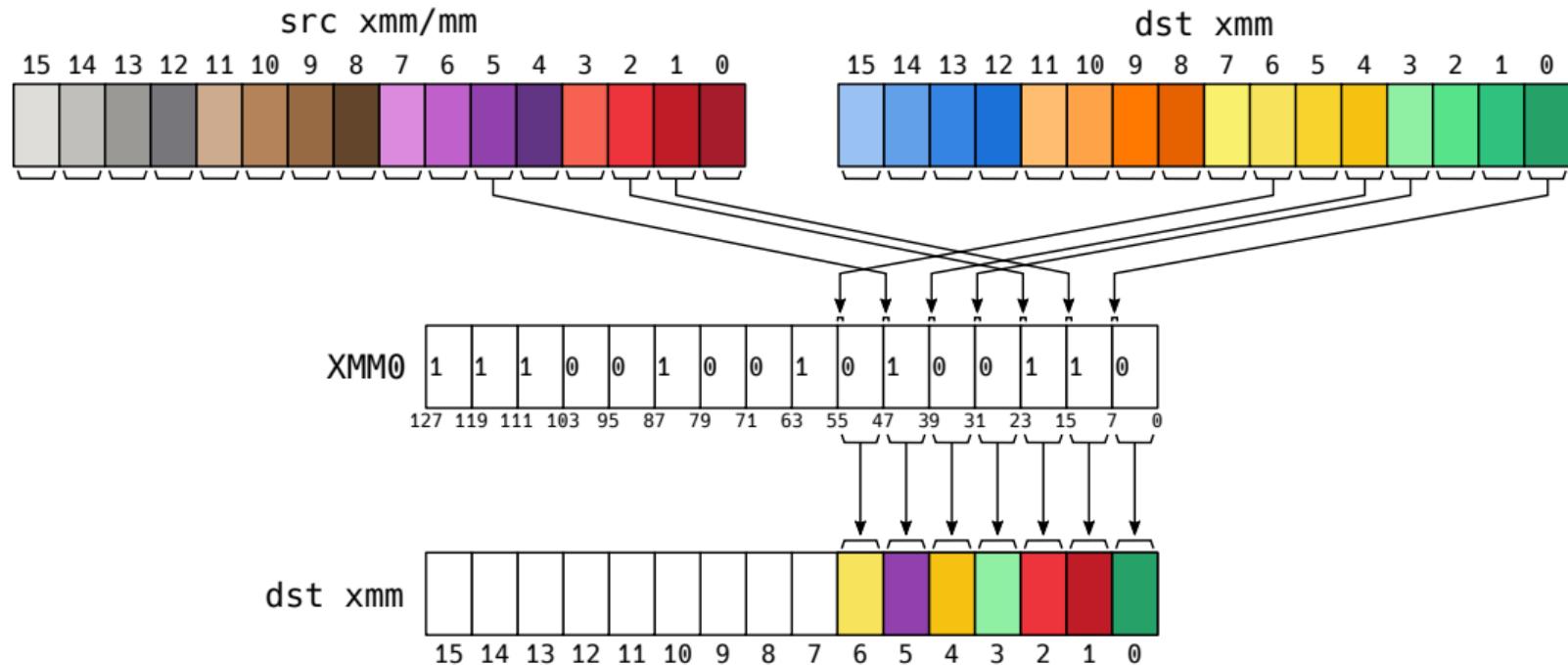
# Ejemplo - PBLENDVB dst, src



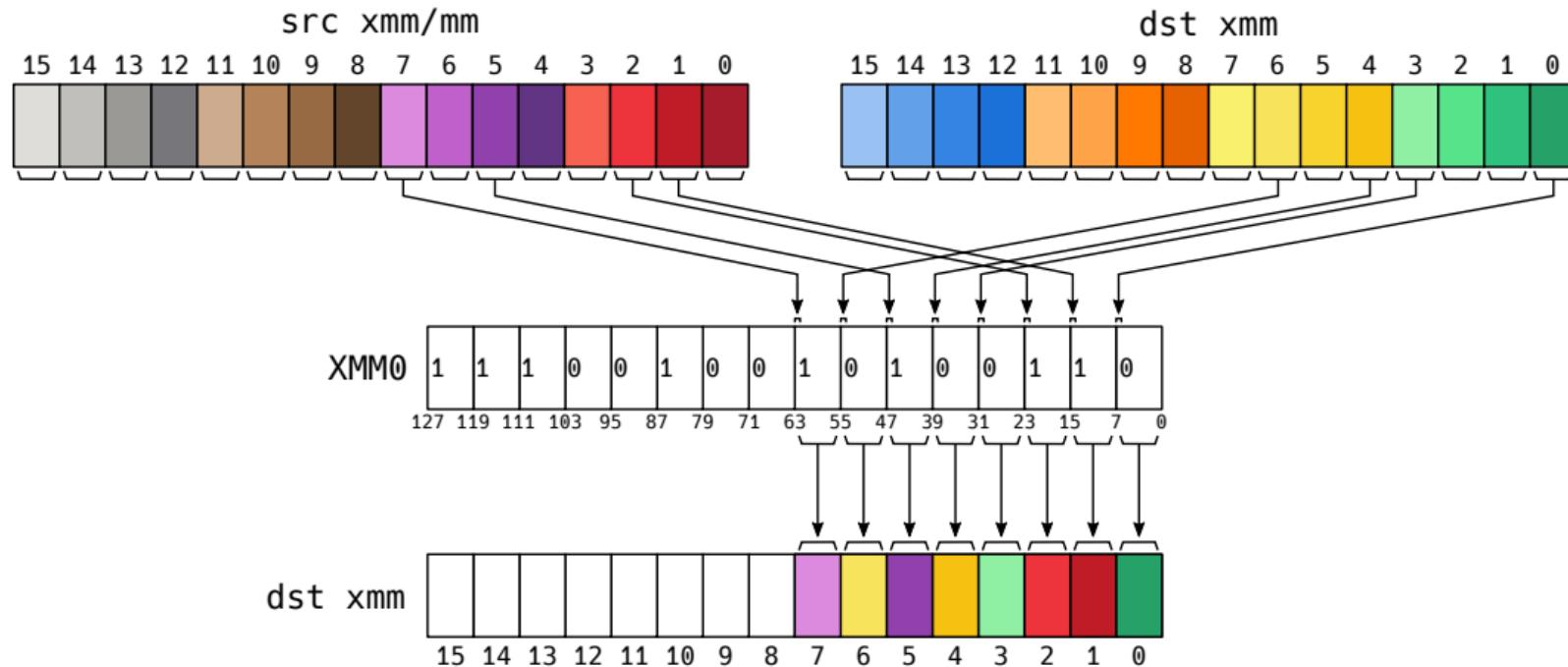
# Ejemplo - PBLENDVB dst, src



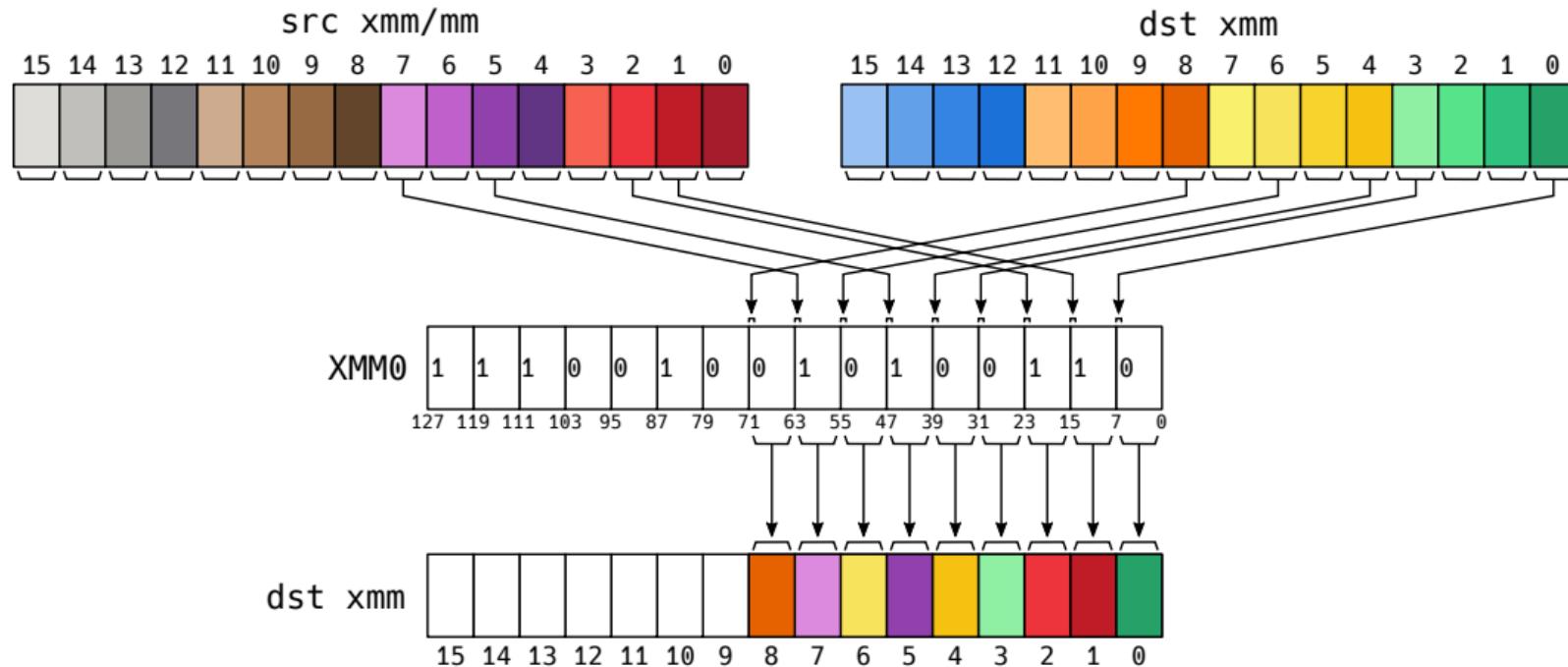
# Ejemplo - PBLENDVB dst, src



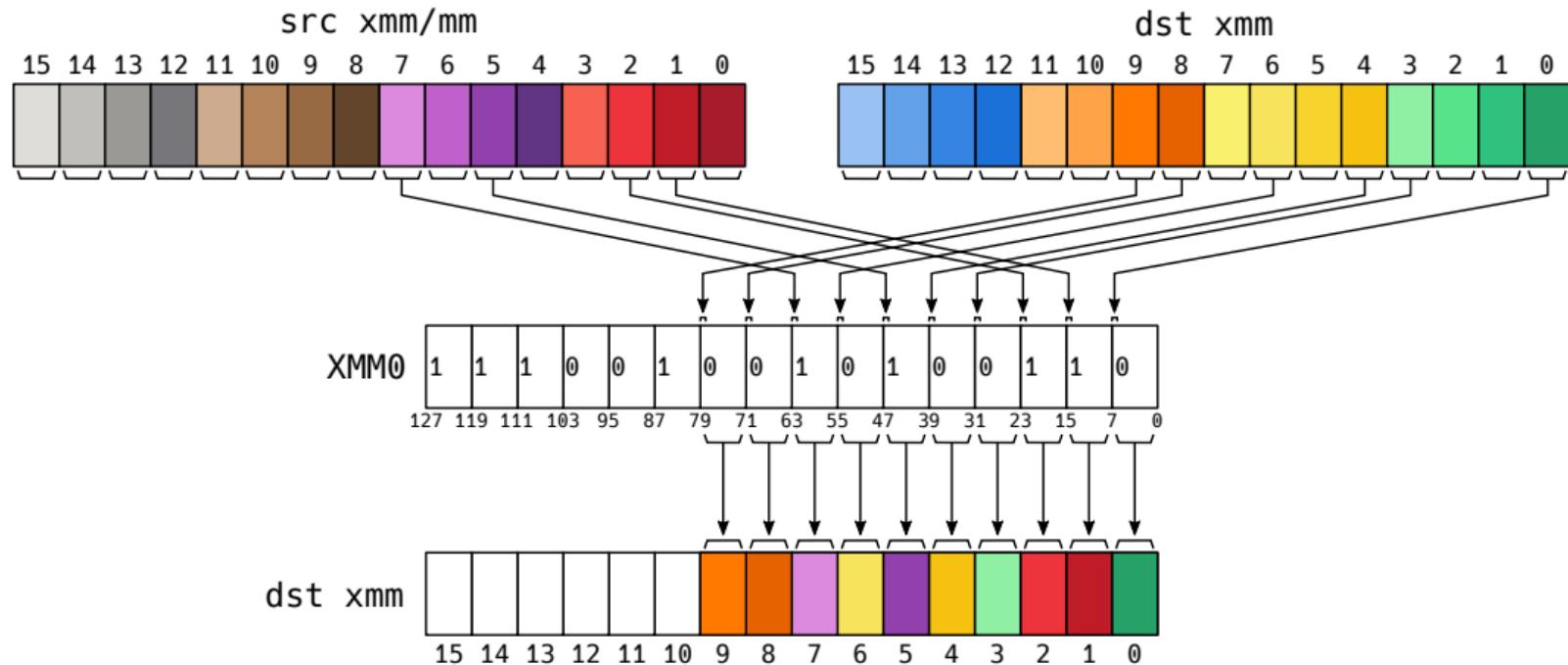
# Ejemplo - PBLENDVB dst, src



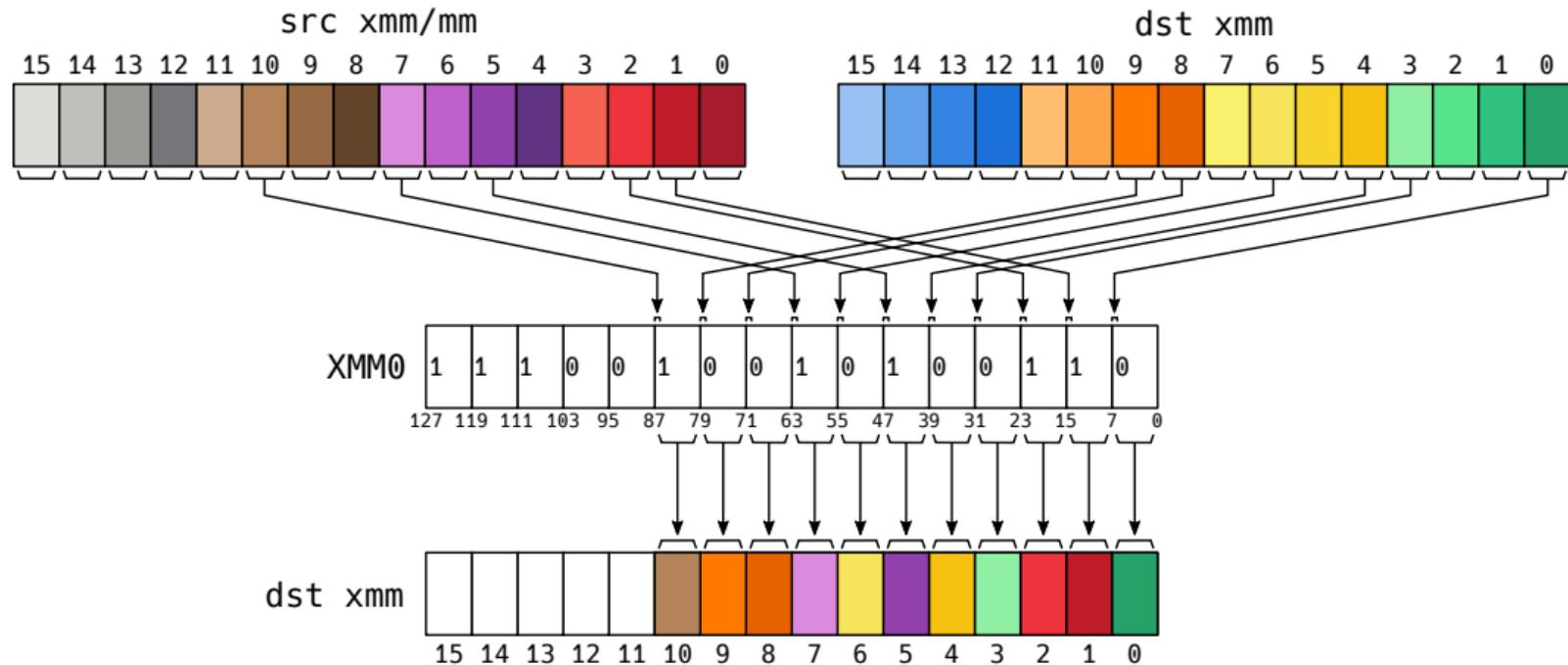
# Ejemplo - PBLENDVB dst, src



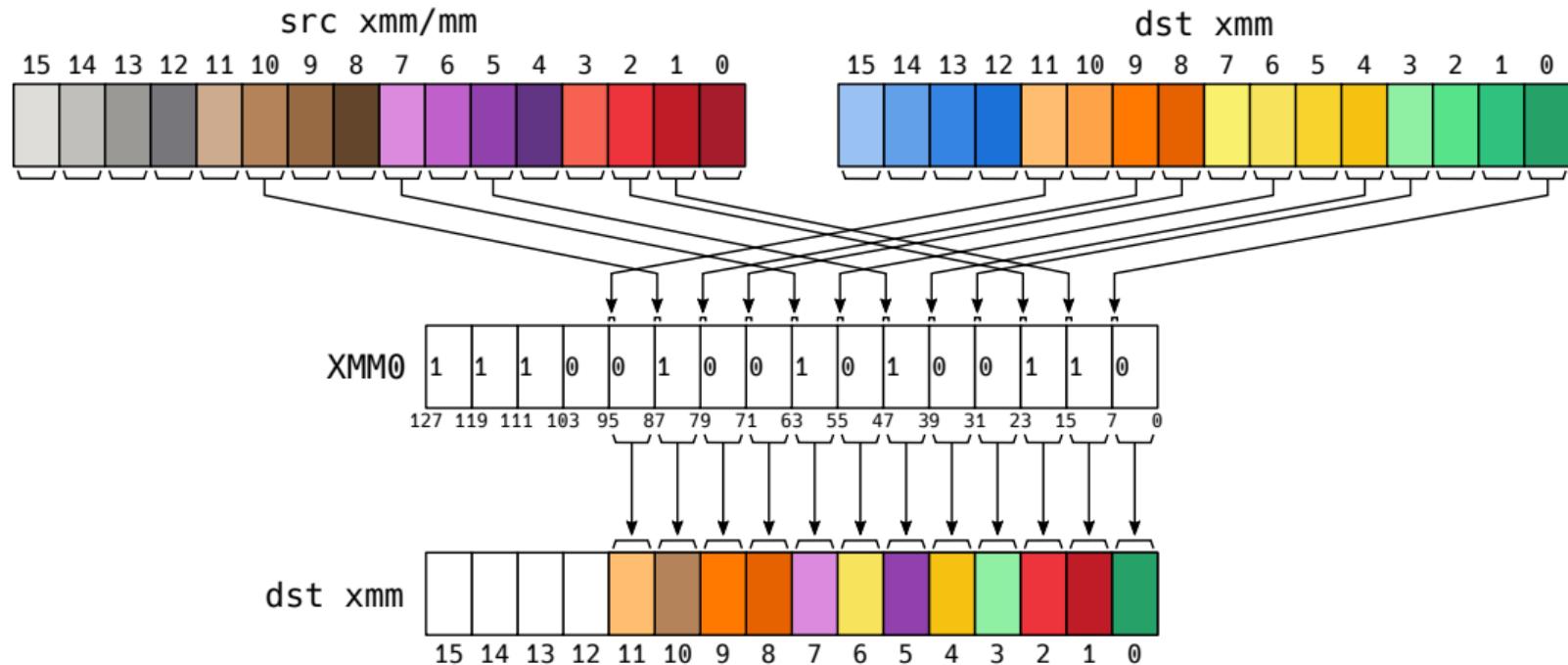
# Ejemplo - PBLENDVB dst, src



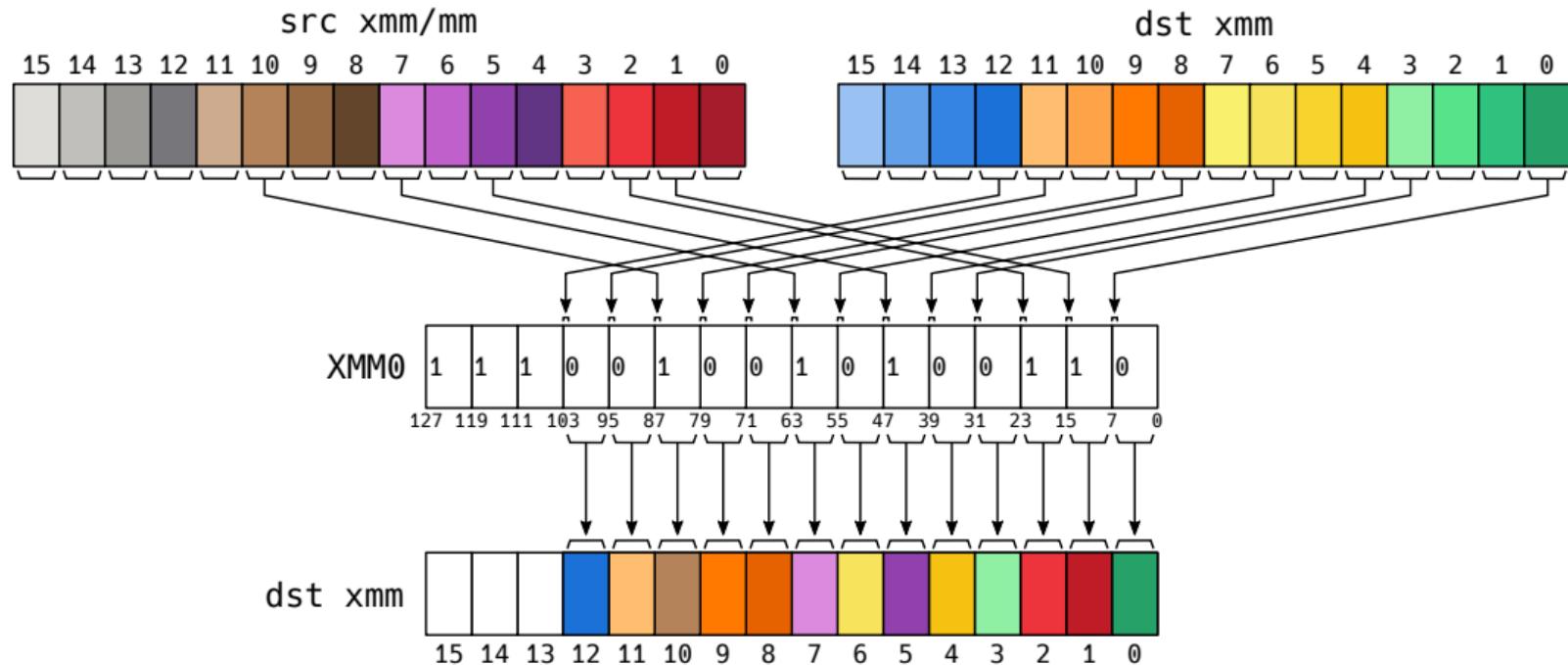
# Ejemplo - PBLENDVB dst, src



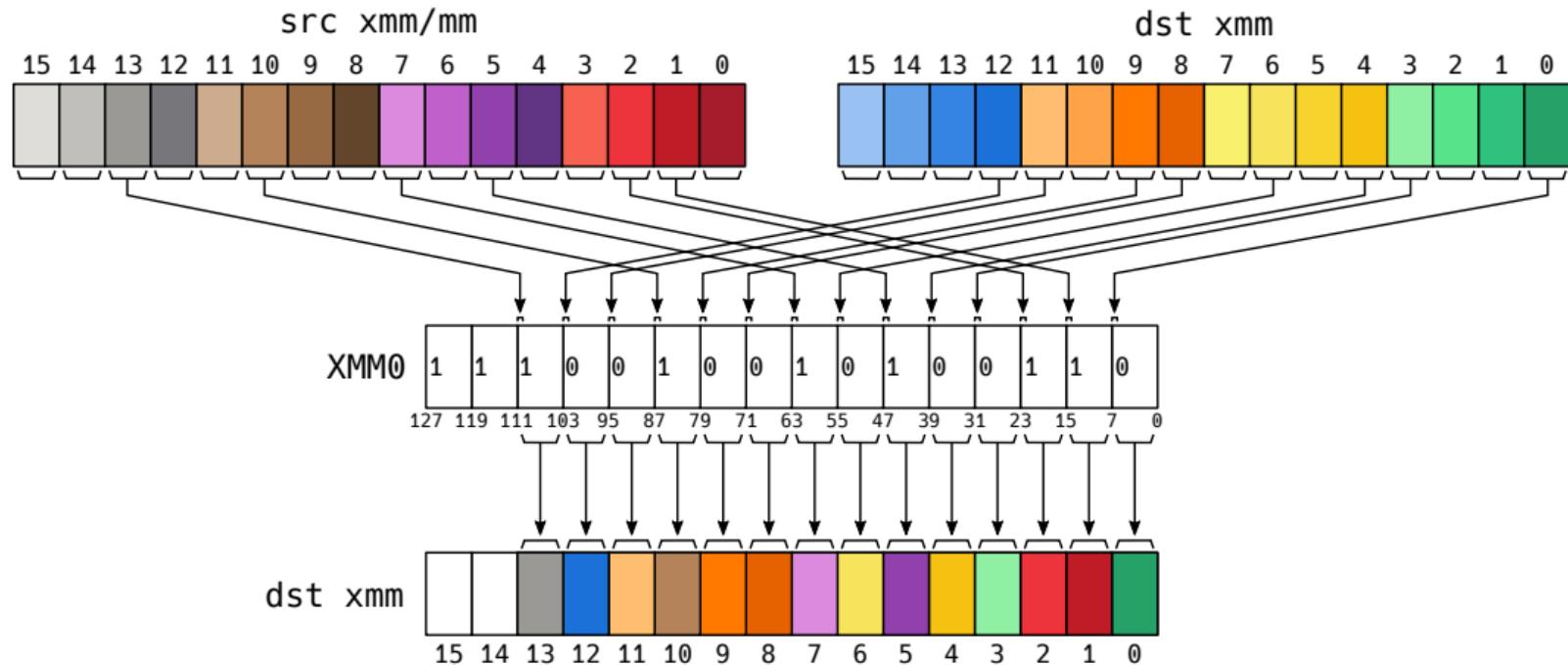
# Ejemplo - PBLENDVB dst, src



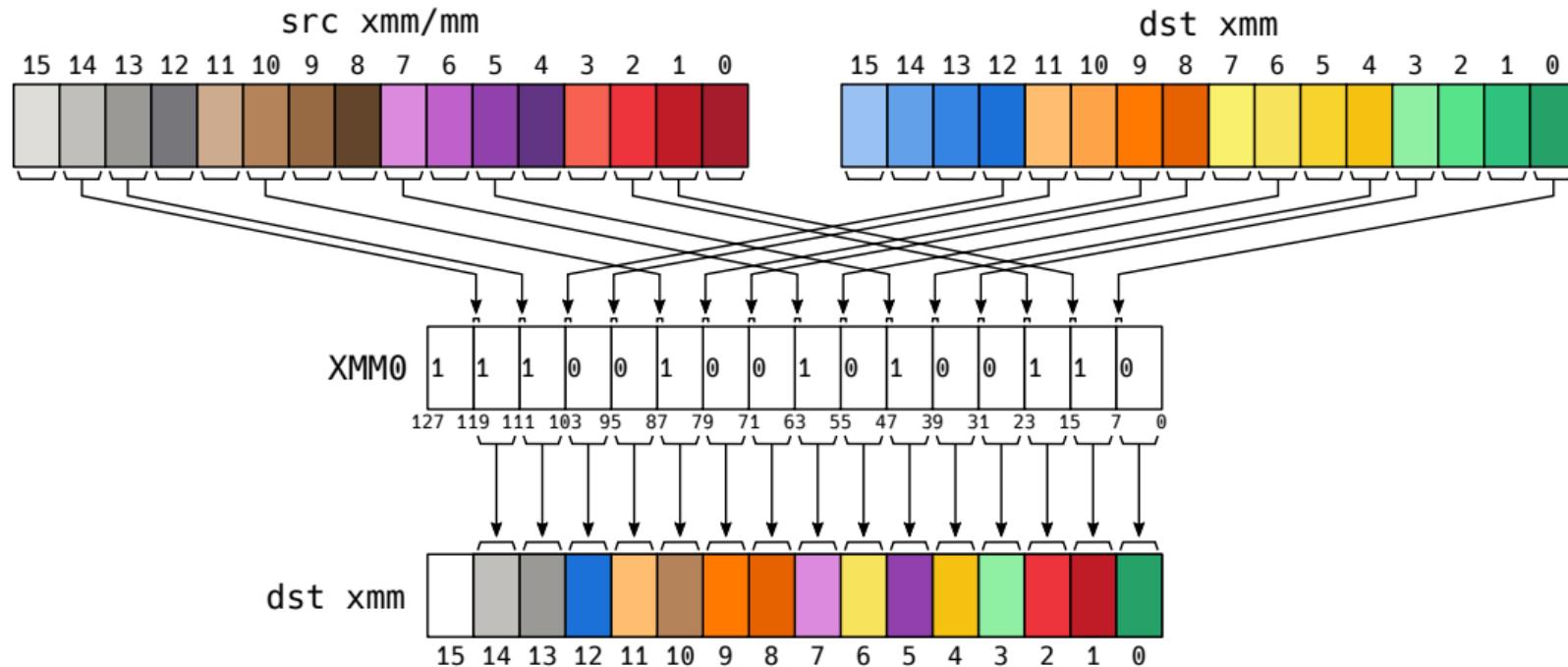
# Ejemplo - PBLENDVB dst, src



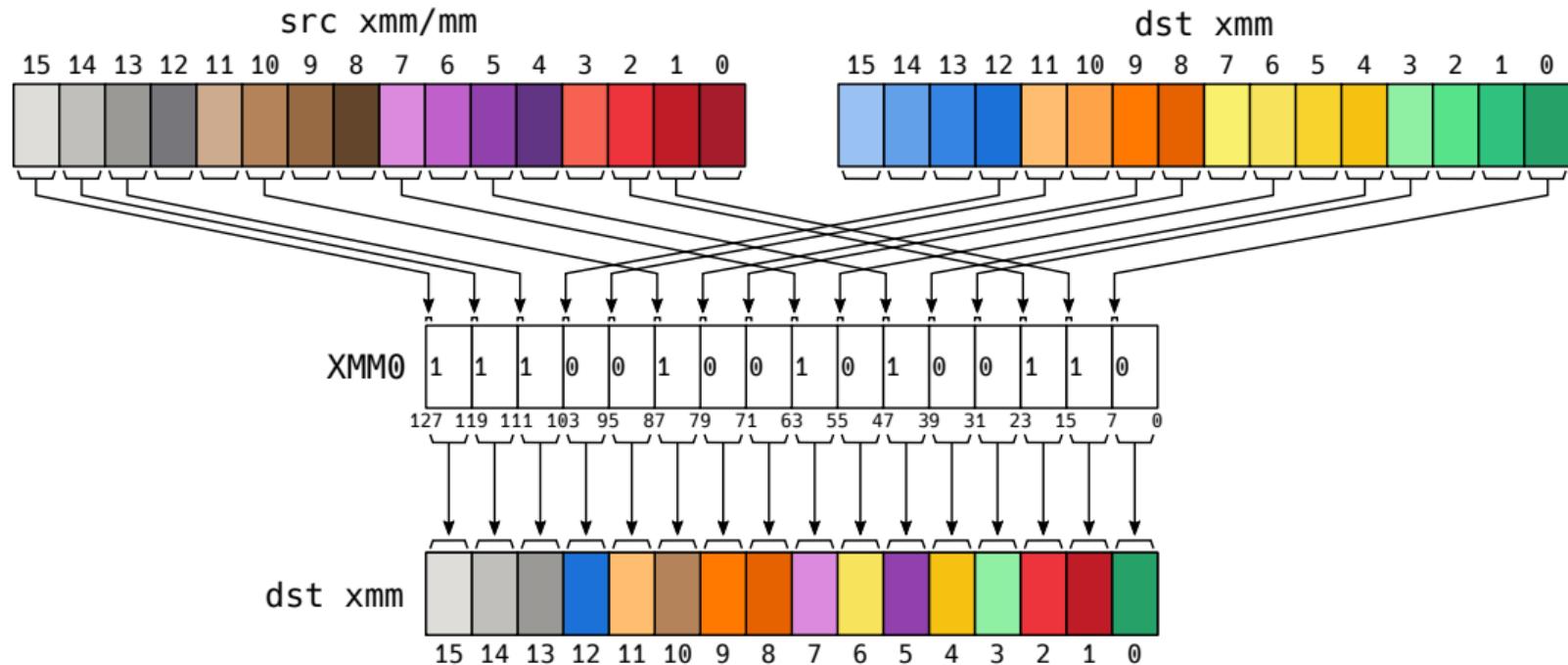
# Ejemplo - PBLENDVB dst, src



# Ejemplo - PBLENDVB dst, src



## Ejemplo - PBLENDVB dst, src



# Conversiones

Las instrucciones de conversión son de la forma: **CVTxx2yy**

Donde **xx** e **yy** pueden valer:

ps - Packed Single FP

ss - Scalar Single FP

pd - Packed Double FP

sd - Scalar Double FP

pi - Packed Integer

si - Scalar Integer

dq - Packed Dword

# Conversiones

Las instrucciones de conversión son de la forma: **CVTxx2yy**

Donde **xx** e **yy** pueden valer:

ps - Packed Single FP	pd - Packed Double FP	pi - Packed Integer
ss - Scalar Single FP	sd - Scalar Double FP	si - Scalar Integer

dq - Packed Dword

## Instrucciones solo de punto flotante

- **CVTSD2SS** - Scalar Double FP to Scalar Single FP (1X) → CVTSD2SS `xmm1, xmm2/m64`
- **CVTSS2SD** - Scalar Single FP to Scalar Double FP (1X) → CVTSS2SD `xmm1, xmm2/m32`

# Conversiones

Las instrucciones de conversión son de la forma: **CVTxx2yy**

Donde **xx** e **yy** pueden valer:

ps - Packed Single FP	pd - Packed Double FP	pi - Packed Integer
ss - Scalar Single FP	sd - Scalar Double FP	si - Scalar Integer

dq - Packed Dword

## Instrucciones solo de punto flotante

- **CVTSD2SS** - Scalar Double FP to Scalar Single FP (1X) → CVTSD2SS `xmm1, xmm2/m64`
- **CVTSS2SD** - Scalar Single FP to Scalar Double FP (1X) → CVTSS2SD `xmm1, xmm2/m32`
- **CVTPD2PS** - Packed Double FP to Packed Single FP (2X) → CVTPD2PS `xmm1, xmm2/m128`
- **CVTPS2PD** - Packed Single FP to Packed Double FP (2X) → CVTPS2PD `xmm1, xmm2/m64`

# Conversiones

## Instrucciones entre enteros y punto flotante

- **CVTSI2SS** - Dword Integer to Scalar Single FP → **CVTSI2SS** xmm, r/m32
- **CVTSS2SI** - Scalar Single FP to Dword Integer → **CVTSS2SI** r32, xmm/m32
- **CVTSI2SD** - Dword Integer to Scalar Double FP → **CVTSI2SD** xmm, r/m64
- **CVTSD2SI** - Scalar Double FP to Dword Integer → **CVTSD2SI** r64, xmm/m64

# Conversiones

## Instrucciones entre enteros y punto flotante

- **CVTSI2SS** - Dword Integer to Scalar Single FP → **CVTSI2SS** `xmm, r/m32`
- **CVTSS2SI** - Scalar Single FP to Dword Integer → **CVTSS2SI** `r32, xmm/m32`
- **CVTSI2SD** - Dword Integer to Scalar Double FP → **CVTSI2SD** `xmm, r/m64`
- **CVTSD2SI** - Scalar Double FP to Dword Integer → **CVTSD2SI** `r64, xmm/m64`
- **CVTDQ2PS** - Packed Dword Integers to Packed Single FP (4X) → **CVTDQ2PS** `xmm1, xmm2/m128`
- **CVTPS2DQ** - Packed Single FP to Packed Dword Integers (4X) → **CVTPS2DQ** `xmm1, xmm2/m128`
- **CVTDQ2PD** - Packed Dword Integers to Packed Double FP (2X) → **CVTDQ2PD** `xmm1, xmm2/m64`
- **CVTPD2DQ** - Packed Double FP to Packed Dword Integers (2X) → **CVTPD2DQ** `xmm1, xmm2/m128`

# Conversiones

## Instrucciones de redondeo

- **ROUNDSS** - Round Scalar Single FP to Integer → `ROUNDSS xmm1, xmm2/m32, imm8`
- **ROUNDSD** - Round Scalar Double FP to Integer → `ROUNDSD xmm1, xmm2/m64, imm8`
- **ROUNDPS** - Round Packed Single FP to Integer (4X) → `ROUNDPS xmm1, xmm2/m128, imm8`
- **ROUNDPD** - Round Packed Double FP to Integer (2X) → `ROUNDPD xmm1, xmm2/m128, imm8`

El parámetro inmediato indica el tipo de redondeo.

# Conversiones

## Instrucciones de redondeo

- **ROUNDSS** - Round Scalar Single FP to Integer → `ROUNDSS xmm1, xmm2/m32, imm8`
- **ROUNDSD** - Round Scalar Double FP to Integer → `ROUNDSD xmm1, xmm2/m64, imm8`
- **ROUNDPS** - Round Packed Single FP to Integer (4X) → `ROUNDPS xmm1, xmm2/m128, imm8`
- **ROUNDPD** - Round Packed Double FP to Integer (2X) → `ROUNDPD xmm1, xmm2/m128, imm8`

El parámetro inmediato indica el tipo de redondeo.

## Instrucciones de truncado

- **CVTTSS2SI** - Truncation Scalar Single FP to Dword Integer (1X) → `CVTTSS2SI r32, xmm/m32`
- **CVTTSDF2SI** - Truncation Scalar Double FP to Signed Integer (1X) → `CVTTSDF2SI r32, xmm/m64`
- **CVTPS2DQ** - Truncation Packed Single FP to Packed Dword Int. (4X) → `CVTPS2DQ xmm1, xmm2/m128`

1 Fundamentos

2 Procesamiento de Señales digitales

3 Modelo de ejecución SIMD

4 Implementaciones SIMD en x86

## 5 Instrucciones

- Transferencias (las mas comunes)
- Aritmética en algoritmos DSP
- Instrucciones de punto flotante
- **Instrucciones para manejo de enteros para SSEn**
- Instrucciones para manejo de cacheabilidad

# Instrucciones de manejo de enteros

- **PAVGB** Calcula el promedio de bytes enteros sin signo empaquetados.
- **PAVGW** Calcula el promedio de words enteras sin signo empaquetadas.
- **PEXTRW** Extrae word.
- **PINSRW** Inserta word.
- **PMAXUB** Máximo de bytes enteros sin signo empaquetados.
- **PMAXSW** Máximo de words enteras signadas empaquetadas.
- **PMINUB** Mínimo de bytes enteros sin signo empaquetados
- **PMINSW** Mínimo de words enteras signadas empaquetadas
- **PMOVMSKB** Mover byte de máscara
- **PMULHUW** Multiplica enteros empaquetados sin signo y almacena el resultado alto
- **PSADBW** Calcula la suma de diferencias absolutas
- **PSHUFW** Cambia de orden words enteras empaquetadas en registros MMX

1 Fundamentos

2 Procesamiento de Señales digitales

3 Modelo de ejecución SIMD

4 Implementaciones SIMD en x86

## 5 Instrucciones

- Transferencias (las mas comunes)
- Aritmética en algoritmos DSP
- Instrucciones de punto flotante
- Instrucciones para manejo de enteros para SSEn
- Instrucciones para manejo de cacheabilidad

# Manejo del cache

- **MASKMOVQ** Almacenamiento No-temporal de los bytes seleccionados de una quadword desde un registro XMM a memoria.
- **MOVNTQ** Almacenamiento No-temporal de una quadword desde un registro XMM a memoria.
- **MOVNTPS** Almacenamiento No-temporal de cuatro valores de punto flotante single-precision desde un registro MMX a memoria.
- **PREFETCHh** Carga 32 bytes (una línea) en el nivel caché especificada a partir de una dirección especificada, hasta un nivel especificado del cache.
- **SFENCE** Serializa operaciones de Store.

1 Fundamentos

2 Procesamiento de Señales digitales

3 Modelo de ejecución SIMD

4 Implementaciones SIMD en x86

5 Instrucciones

6 Advanced Vector Extensions

- Preliminar

- Detección de las facilidades AVX
- Soporte a Half Precision
- Soporte para fusión de productos y sumas

# Características arquitecturales

# Características arquitecturales

- Capacidad de procesamiento de vectores de 256 bit.

# Características arquitecturales

- Capacidad de procesamiento de vectores de 256 bit.
- Extiende el set de instrucciones vectoriales de 128 bit mediante un sistema de codificación basado en un prefijo de instrucción **VEX**: Vector Extended Prefix.

# Características arquitecturales

- Capacidad de procesamiento de vectores de 256 bit.
- Extiende el set de instrucciones vectoriales de 128 bit mediante un sistema de codificación basado en un prefijo de instrucción **VEX**: Vector Extended Prefix.
- **FMA (Fused Multiply Add)**: Mejoras al cálculo de punto flotante empaquetado en vectores de 128 bit, introduciendo instrucciones que fusionan producto y sumas, producto y restas, producto y suma y resta entrelazadas, e inversión de signo en fusión de producto y suma y fusión de producto y resta.

# Características arquitecturales

- Capacidad de procesamiento de vectores de 256 bit.
- Extiende el set de instrucciones vectoriales de 128 bit mediante un sistema de codificación basado en un prefijo de instrucción **VEX**: Vector Extended Prefix.
- **FMA** (Fused Multiply Add): Mejoras al cálculo de punto flotante empaquetado en vectores de 128 bit, introduciendo instrucciones que fusionan producto y sumas, producto y restas, producto y suma y resta entrelazadas, e inversión de signo en fusión de producto y suma y fusión de producto y resta.
- Posteriormente **AVX2** incorpora mas instrucciones para optimizar cálculo no solo en vectores de punto flotante de 256 bit sino en vectores de enteros de 256 bit.

# Características arquitecturales

- Capacidad de procesamiento de vectores de 256 bit.
- Extiende el set de instrucciones vectoriales de 128 bit mediante un sistema de codificación basado en un prefijo de instrucción **VEX**: Vector Extended Prefix.
- **FMA** (Fused Multiply Add): Mejoras al cálculo de punto flotante empaquetado en vectores de 128 bit, introduciendo instrucciones que fusionan producto y sumas, producto y restas, producto y suma y resta entrelazadas, e inversión de signo en fusión de producto y suma y fusión de producto y resta.
- Posteriormente **AVX2** incorpora mas instrucciones para optimizar cálculo no solo en vectores de punto flotante de 256 bit sino en vectores de enteros de 256 bit.
- Incorpora instrucciones con tres operandos para simplificar la vectorización en compiladores en instrucciones de alto nivel.

# Características arquitecturales

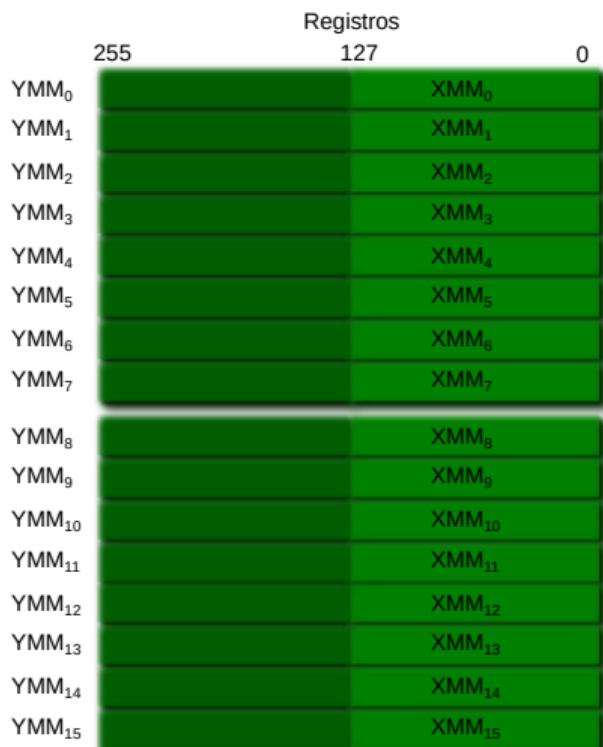
Registros		
255	127	0
YMM <sub>0</sub>	XMM <sub>0</sub>	
YMM <sub>1</sub>	XMM <sub>1</sub>	
YMM <sub>2</sub>	XMM <sub>2</sub>	
YMM <sub>3</sub>	XMM <sub>3</sub>	
YMM <sub>4</sub>	XMM <sub>4</sub>	
YMM <sub>5</sub>	XMM <sub>5</sub>	
YMM <sub>6</sub>	XMM <sub>6</sub>	
YMM <sub>7</sub>	XMM <sub>7</sub>	
YMM <sub>8</sub>	XMM <sub>8</sub>	
YMM <sub>9</sub>	XMM <sub>9</sub>	
YMM <sub>10</sub>	XMM <sub>10</sub>	
YMM <sub>11</sub>	XMM <sub>11</sub>	
YMM <sub>12</sub>	XMM <sub>12</sub>	
YMM <sub>13</sub>	XMM <sub>13</sub>	
YMM <sub>14</sub>	XMM <sub>14</sub>	
YMM <sub>15</sub>	XMM <sub>15</sub>	

# Características arquitecturales

Registros		
255	127	0
YMM <sub>0</sub>	XMM <sub>0</sub>	
YMM <sub>1</sub>	XMM <sub>1</sub>	
YMM <sub>2</sub>	XMM <sub>2</sub>	
YMM <sub>3</sub>	XMM <sub>3</sub>	
YMM <sub>4</sub>	XMM <sub>4</sub>	
YMM <sub>5</sub>	XMM <sub>5</sub>	
YMM <sub>6</sub>	XMM <sub>6</sub>	
YMM <sub>7</sub>	XMM <sub>7</sub>	
YMM <sub>8</sub>	XMM <sub>8</sub>	
YMM <sub>9</sub>	XMM <sub>9</sub>	
YMM <sub>10</sub>	XMM <sub>10</sub>	
YMM <sub>11</sub>	XMM <sub>11</sub>	
YMM <sub>12</sub>	XMM <sub>12</sub>	
YMM <sub>13</sub>	XMM <sub>13</sub>	
YMM <sub>14</sub>	XMM <sub>14</sub>	
YMM <sub>15</sub>	XMM <sub>15</sub>	

- Soporta vectores de 256 bit en los registros YMM.

# Características arquitecturales



- Soporta vectores de 256 bit en los registros **YMM**.
- Acelera el cálculo en punto flotante hasta 2 veces respecto de las extensiones **SSE** que emplean registros de 128 bit.

# Características arquitecturales

Registros		
255	127	0
YMM <sub>0</sub>	XMM <sub>0</sub>	
YMM <sub>1</sub>	XMM <sub>1</sub>	
YMM <sub>2</sub>	XMM <sub>2</sub>	
YMM <sub>3</sub>	XMM <sub>3</sub>	
YMM <sub>4</sub>	XMM <sub>4</sub>	
YMM <sub>5</sub>	XMM <sub>5</sub>	
YMM <sub>6</sub>	XMM <sub>6</sub>	
YMM <sub>7</sub>	XMM <sub>7</sub>	
YMM <sub>8</sub>	XMM <sub>8</sub>	
YMM <sub>9</sub>	XMM <sub>9</sub>	
YMM <sub>10</sub>	XMM <sub>10</sub>	
YMM <sub>11</sub>	XMM <sub>11</sub>	
YMM <sub>12</sub>	XMM <sub>12</sub>	
YMM <sub>13</sub>	XMM <sub>13</sub>	
YMM <sub>14</sub>	XMM <sub>14</sub>	
YMM <sub>15</sub>	XMM <sub>15</sub>	

- Soporta vectores de 256 bit en los registros **YMM**.
- Acelera el cálculo en punto flotante hasta 2 veces respecto de las extensiones **SSE** que emplean registros de 128 bit.
- Los Registros **YMM<sub>0</sub>** – **YMM<sub>7</sub>** se emplean en todos los modos de trabajo, mientras que **YMM<sub>8</sub>** – **YMM<sub>15</sub>** se agregan en el modo IA32e de 64 bit.

# Características arquitecturales

Registros		
255	127	0
YMM <sub>0</sub>	XMM <sub>0</sub>	
YMM <sub>1</sub>	XMM <sub>1</sub>	
YMM <sub>2</sub>	XMM <sub>2</sub>	
YMM <sub>3</sub>	XMM <sub>3</sub>	
YMM <sub>4</sub>	XMM <sub>4</sub>	
YMM <sub>5</sub>	XMM <sub>5</sub>	
YMM <sub>6</sub>	XMM <sub>6</sub>	
YMM <sub>7</sub>	XMM <sub>7</sub>	
YMM <sub>8</sub>	XMM <sub>8</sub>	
YMM <sub>9</sub>	XMM <sub>9</sub>	
YMM <sub>10</sub>	XMM <sub>10</sub>	
YMM <sub>11</sub>	XMM <sub>11</sub>	
YMM <sub>12</sub>	XMM <sub>12</sub>	
YMM <sub>13</sub>	XMM <sub>13</sub>	
YMM <sub>14</sub>	XMM <sub>14</sub>	
YMM <sub>15</sub>	XMM <sub>15</sub>	

- Soporta vectores de 256 bit en los registros **YMM**.
- Acelera el cálculo en punto flotante hasta 2 veces respecto de las extensiones **SSE** que emplean registros de 128 bit.
- Los Registros **YMM<sub>0</sub>** – **YMM<sub>7</sub>** se emplean en todos los modos de trabajo, mientras que **YMM<sub>8</sub>** – **YMM<sub>15</sub>** se agregan en el modo IA32e de 64 bit.
- La parte baja de los registros **YMM**, es un alias de los registros **XMM** del modo **SSE**.

# Mejoras en la sintaxis

# Mejoras en la sintaxis

- El tercer operando (operandos no destructivos), reduce la cantidad de instrucciones eliminando transferencias registro-registro anteriores y/o posteriores a la instrucción de interés. Esto reduce significativamente la ejecución de loops, reduce el tamaño del código y mejora las chances de microfusión de micro-operaciones.

## Mejoras en la sintaxis

- El tercer operando (operandos no destructivos), reduce la cantidad de instrucciones eliminando transferencias registro-registro anteriores y/o posteriores a la instrucción de interés. Esto reduce significativamente la ejecución de loops, reduce el tamaño del código y mejora las chances de microfusión de micro-operaciones.
- Tercer operando fuente, codificado en los 4 bit superiores de un operando inmediato de 8 bit. Solo en un pequeño grupo de instrucciones de cuatro operandos

# Mejoras en la sintaxis

- El tercer operando (operandos no destructivos), reduce la cantidad de instrucciones eliminando transferencias registro-registro anteriores y/o posteriores a la instrucción de interés. Esto reduce significativamente la ejecución de loops, reduce el tamaño del código y mejora las chances de microfusión de micro-operaciones.
- Tercer operando fuente, codificado en los 4 bit superiores de un operando inmediato de 8 bit. Solo en un pequeño grupo de instrucciones de cuatro operandos

```
MOVSD xmm1, xmm2 ; Copia un operando a xmm1
```

```
ADDPD xmm1, xmm3/m128 ; Suma de dos operandos en SSE
```

```
VADDPS xmm1, xmm2, xmm3/m128 ; Equivalente AVX con 3 operandos.
```

; Trabaja en 128 bit.

; Los bit YMM255-YMM128 se ponen

; automáticamente en 0.

# Prefijo VEX

# Prefijo VEX

- Las extensiones **AVX** introducen un nuevo prefijo: **VEX**, tanto en el formato de codificación de instrucciones de 64 bit como 32 bit.

# Prefijo VEX

- Las extensiones **AVX** introducen un nuevo prefijo: **VEX**, tanto en el formato de codificación de instrucciones de 64 bit como 32 bit.
- Permite implementar instrucciones de tres operandos (operando no destructivo), cuando actúa como Prefijo de Registro.

# Prefijo VEX

- Las extensiones **AVX** introducen un nuevo prefijo: **VEX**, tanto en el formato de codificación de instrucciones de 64 bit como 32 bit.
- Permite implementar instrucciones de tres operandos (operando no destructivo), cuando actúa como Prefijo de Registro.
- Optimiza la codificación de instrucciones SIMD sobre datasets de 128 bit y 256 bit.

# Prefijo VEX

- Las extensiones **AVX** introducen un nuevo prefijo: **VEX**, tanto en el formato de codificación de instrucciones de 64 bit como 32 bit.
- Permite implementar instrucciones de tres operandos (operando no destructivo), cuando actúa como Prefijo de Registro.
- Optimiza la codificación de instrucciones SIMD sobre datasets de 128 bit y 256 bit.
- Absorbe la funcionalidad del prefijo **REX** introducido con el modo 64 bit. De hecho si codificamos los dos prefijos en una misma instrucción se genera una excepción **#UD**.

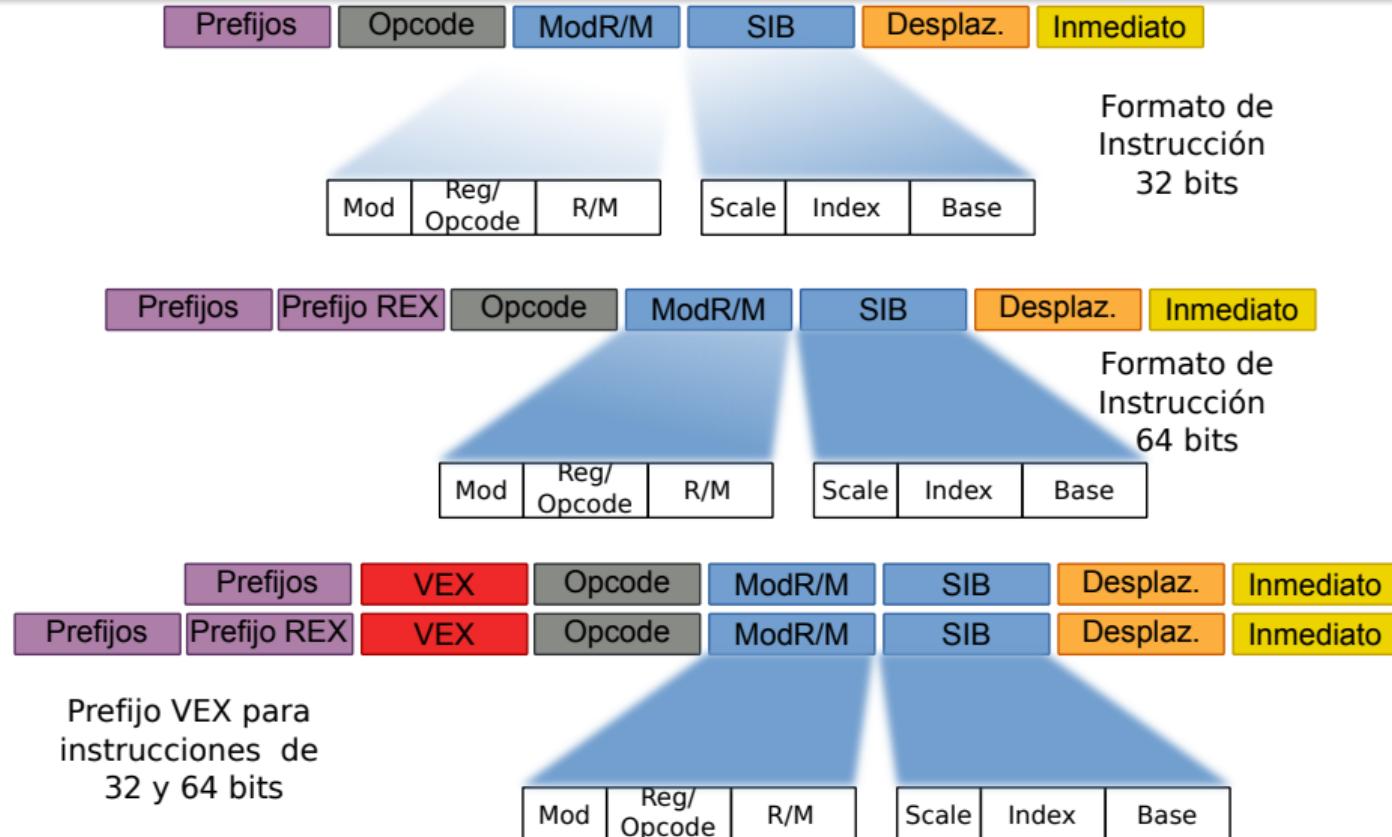
# Prefijo VEX

- Las extensiones **AVX** introducen un nuevo prefijo: **VEX**, tanto en el formato de codificación de instrucciones de 64 bit como 32 bit.
- Permite implementar instrucciones de tres operandos (operando no destructivo), cuando actúa como Prefijo de Registro.
- Optimiza la codificación de instrucciones SIMD sobre datasets de 128 bit y 256 bit.
- Absorbe la funcionalidad del prefijo **REX** introducido con el modo 64 bit. De hecho si codificamos los dos prefijos en una misma instrucción se genera una excepción **#UD**.
- También absorbe la funcionalidad de los prefijos SIMD pre-existentes (0x66, 0xF2, y 0xF3), y de los opcode de Escape de un byte (0x0F), y dos bytes (0xF38 y 0xF3A), y del prefijo **LOCK**. Estas combinaciones también generarán **#UD**.

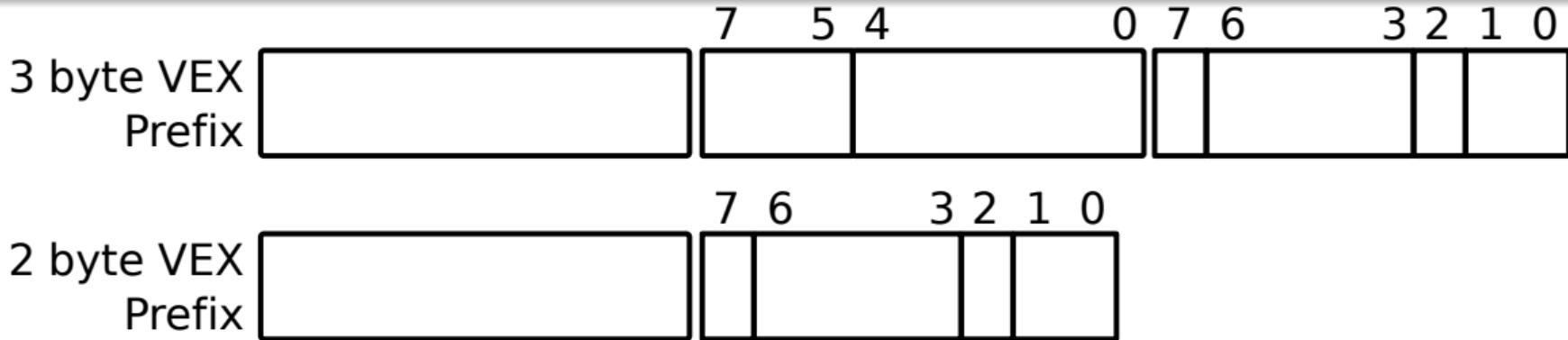
# Prefijo VEX

- Las extensiones **AVX** introducen un nuevo prefijo: **VEX**, tanto en el formato de codificación de instrucciones de 64 bit como 32 bit.
- Permite implementar instrucciones de tres operandos (operando no destructivo), cuando actúa como Prefijo de Registro.
- Optimiza la codificación de instrucciones SIMD sobre datasets de 128 bit y 256 bit.
- Absorbe la funcionalidad del prefijo **REX** introducido con el modo 64 bit. De hecho si codificamos los dos prefijos en una misma instrucción se genera una excepción **#UD**.
- También absorbe la funcionalidad de los prefijos SIMD pre-existentes (0x66, 0xF2, y 0xF3), y de los opcode de Escape de un byte (0x0F), y dos bytes (0xF38 y 0xF3A), y del prefijo **LOCK**. Estas combinaciones también generarán **#UD**.
- Cuando se emplean los prefijos SIMD relaja los requerimientos de alineación en memoria de los operandos.

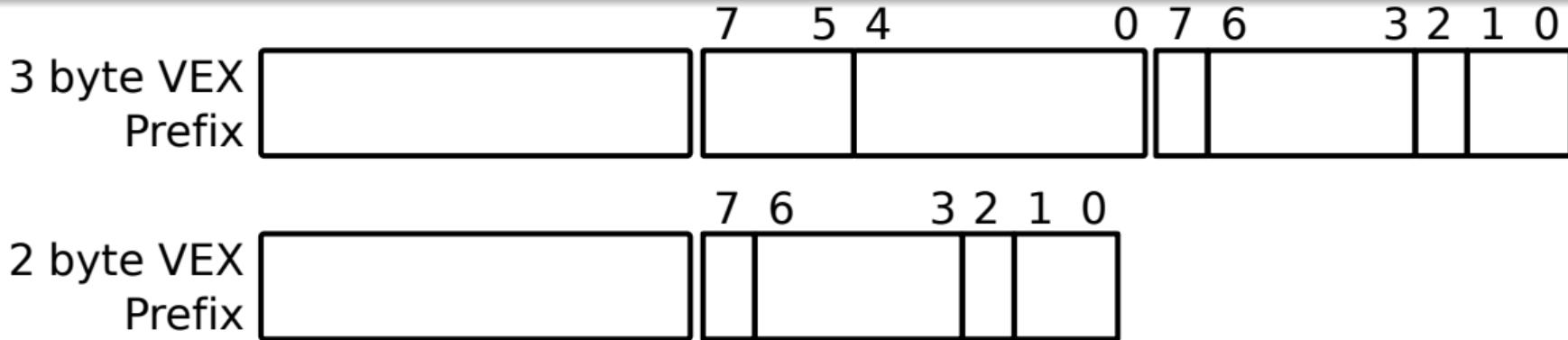
# Formato de instrucciones



# Prefijo VEX

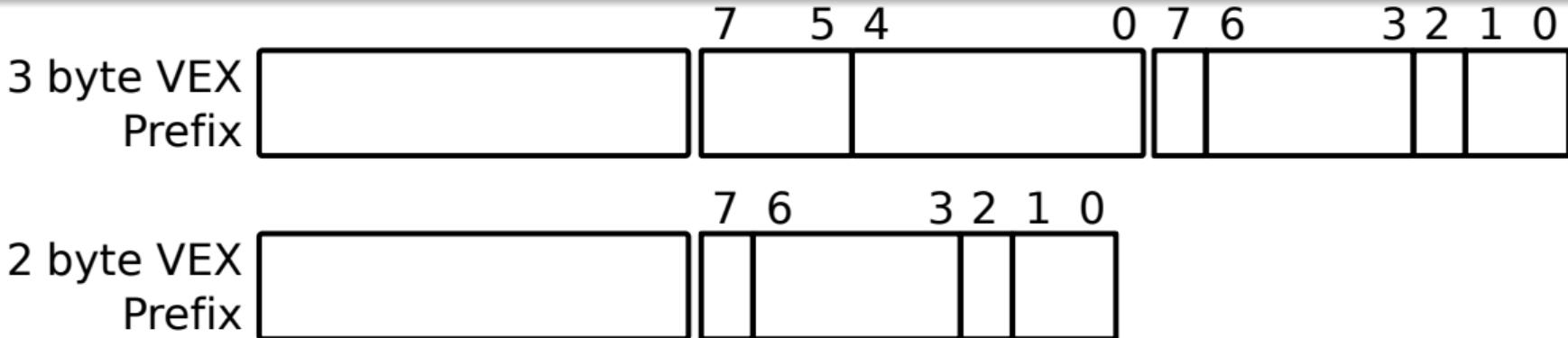


# Prefijo VEX



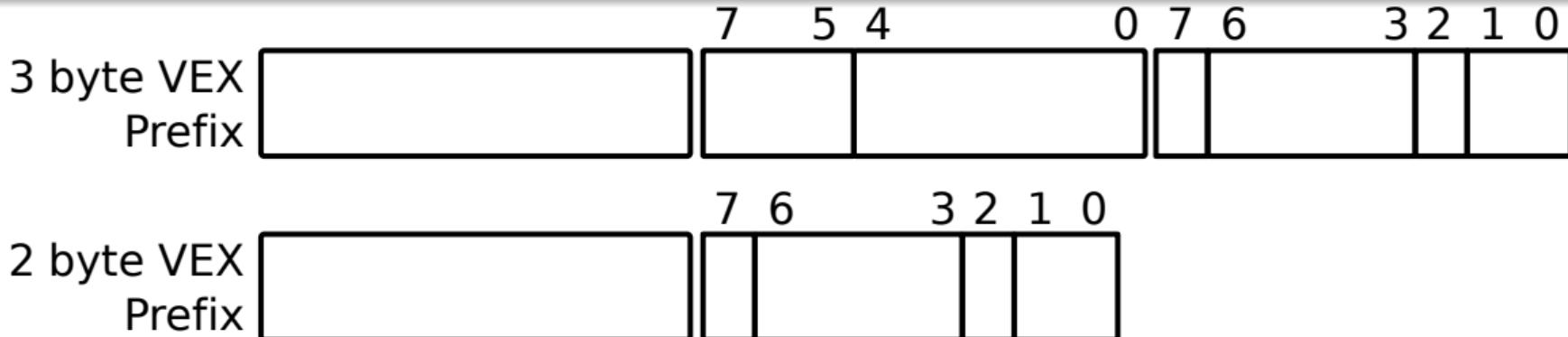
- Dos versiones: 2 B y 3 B.

# Prefijo VEX



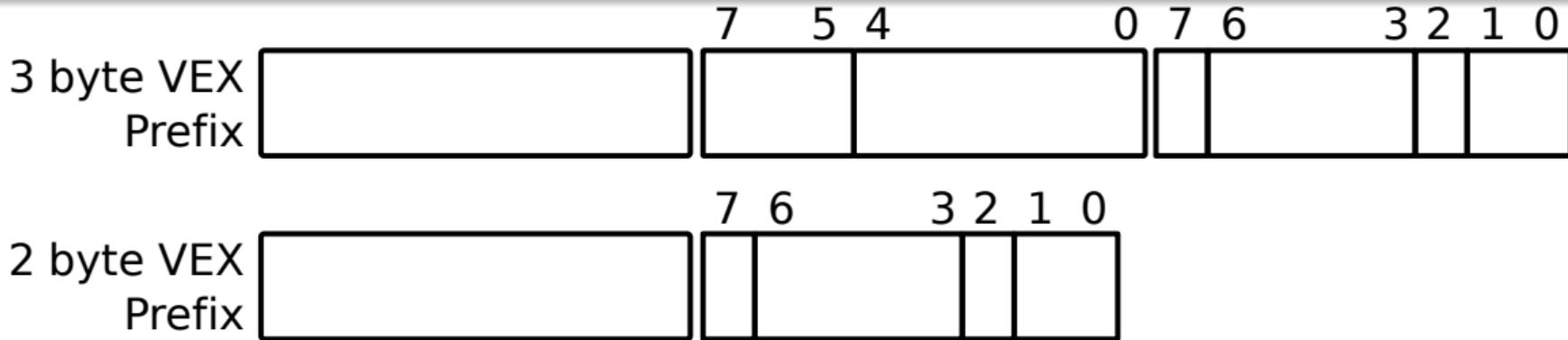
- Dos versiones: 2 B y 3 B.
- El de 2 B se emplea mayoritariamente para operaciones con escalares, para adaptar operaciones sobre registros legacy de 128 bit, y para las operaciones de 256 bit

# Prefijo VEX

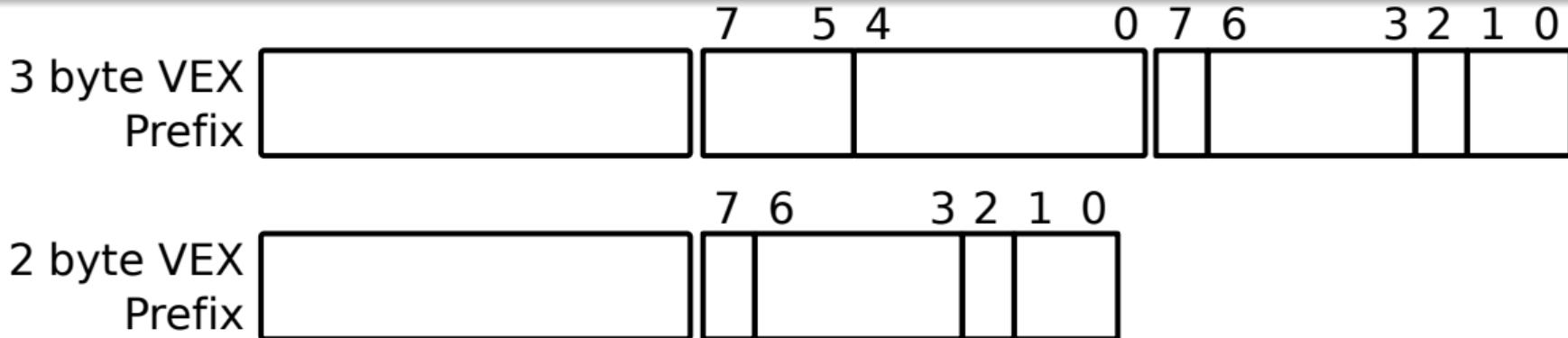


- Dos versiones: 2 B y 3 B.
- El de 2 B se emplea mayoritariamente para operaciones con escalares, para adaptar operaciones sobre registros legacy de 128 bit, y para las operaciones de 256 bit
- El prefijo de 3 B, provee una forma robusta de reemplazar al prefijo **REX**, permite instrucciones de tres operandos, incluyendo instrucciones **AVX** y **FMA**.

# Prefijo VEX

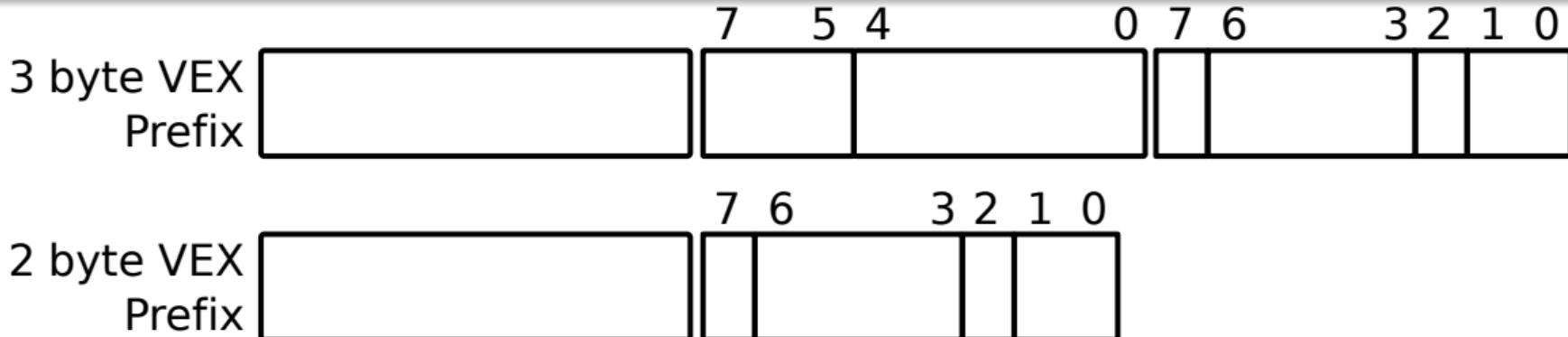


## Prefijo VEX



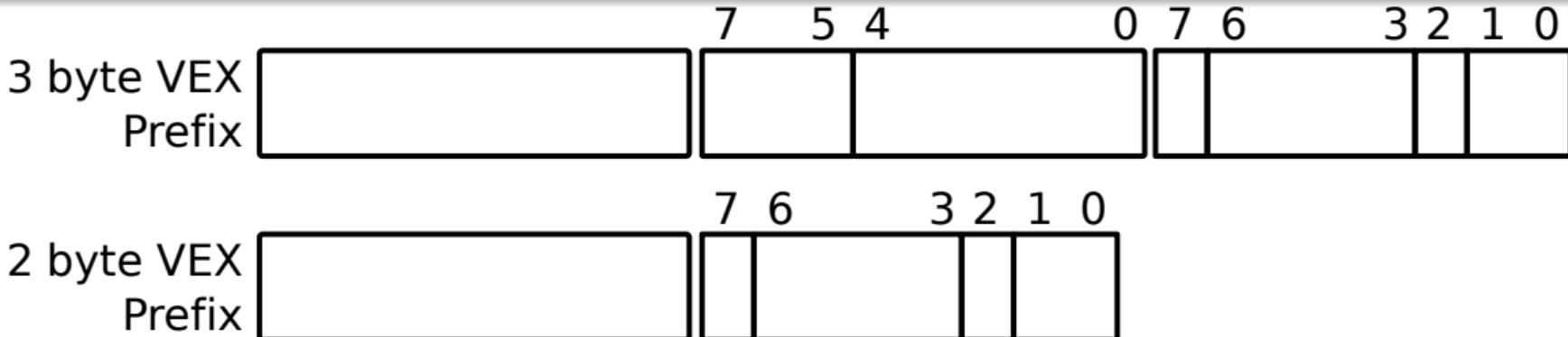
- Para implementar el operando no destructivo (3er. operando fuente), se usa el campo **REX.vvvv**. El código es invertido, es decir que 1111b corresponde a los registros **ymm0**, **xmm0**, o **r0**, y 0000b corresponde a los registros **ymm15**, **xmm15**, o **r15**.

## Prefijo VEX



- Para implementar el operando no destructivo (3er. operando fuente), se usa el campo **REX.vvvv**. El código es invertido, es decir que 1111b corresponde a los registros **ymm0**, **xmm0**, o **r0**, y 0000b corresponde a los registros **ymm15**, **xmm15**, o **r15**.
- VEX.R**, con la misma lógica inversa reemplaza la función de **REX.R**, que permite modificar el campo **SIB** para seleccionar los registros **r8 - r15**

# Prefijo VEX



- Para implementar el operando no destructivo (3er. operando fuente), se usa el campo **REX.vvvv**. El código es invertido, es decir que 1111b corresponde a los registros **ymm0**, **xmm0**, o **r0**, y 0000b corresponde a los registros **ymm15**, **xmm15**, o **r15**.
- VEX.R**, con la misma lógica inversa reemplaza la función de **REX.R**, que permite modificar el campo **SIB** para seleccionar los registros **r8 - r15**
- VEX.X** y **VEX.B**, con la misma lógica inversa reemplazan a **REX.X** y **REX.B**, que modifican el campo **SIB**, ampliando las capacidades de operandos en modo 64 bit.

1 Fundamentos

2 Procesamiento de Señales digitales

3 Modelo de ejecución SIMD

4 Implementaciones SIMD en x86

5 Instrucciones

6 Advanced Vector Extensions

- Preliminar
- Detección de las facilidades AVX
- Soporte a Half Precision
- Soporte para fusión de productos y sumas

# Detección de soporte a AVX

```
supports_AVX:  
    mov eax, 1  
    cpuid  
    and ecx, 018000000H  
    cmp ecx, 018000000H; checkea flags OSXSAVE y AVX  
    jne not_supported  
; El procesador soporta AVX y XGETBV fue habilitada por el OS  
    mov ecx, 0; ecx=0 para leer registro XCRO  
    XGETBV ; EDX:EAX = XCRO  
    and eax, 06H  
    cmp eax, 06H; habilitó el OS el soporte de estado de XMM y YMM?  
    jne not_supported  
    mov eax, 1  
    jmp done  
not_supported:  
    mov eax, 0  
done:  
    ret
```

# Soporte a instrucciones AESNI codificadas para VEX

Las instrucciones **VAESDEC VAESDECLAST VAESENC VAESENCLAST VAESIMC VAESKEYGENASSIST**, operan en base a estados de los registros **ymm**

```
supports_VAESNI:  
    mov eax, 1  
    cpuid  
    and ecx, 01A000000H  
    cmp ecx, 01A000000H; checkea flags OSXSAVE AVX y AESNI.  
    jne not_supported  
; El Procesador soporta AVX, VEX-encoded AESNI, y el OS habilitó XGETBV.  
    mov ecx, 0; ecx=0 para leer registro XCRO  
    XGETBV ; EDX:EAX = XCRO  
    and eax, 06H  
    cmp eax, 06H; habilitó el OS el soporte de estado de XMM y YMM?  
    jne not_supported  
    mov eax, 1  
    jmp done  
not_supported:  
    mov eax, 0  
done:  
    ret
```

# Detección de soporte instrucción VPCLMULQDQ

Similarmente la instrucción **VPCLMULQDQ**, requiere comprobar estar soportada por el procesador.

```
supports_VPCLMULQDQ:  
    mov eax, 1  
    cpuid  
    and ecx, 018000002H  
    cmp ecx, 018000002H; checkea flags OSXSAVE AVX y PCLMULQDQ.  
    jne not_supported  
; El Procesador soporta AVX, VEX-encoded PCLMULQDQ, y el OS habilitó XGETBV.  
    mov ecx, 0; ecx=0 para leer registro XCRO  
    XGETBV ; EDX:EAX = XCRO  
    and eax, 06H  
    cmp eax, 06H; habilitó el OS el soporte de estado para XMM y YMM?  
    jne not_supported  
    mov eax, 1  
    jmp done  
not_supported:  
    mov eax, 0  
done:  
    ret
```

1 Fundamentos

2 Procesamiento de Señales digitales

3 Modelo de ejecución SIMD

4 Implementaciones SIMD en x86

5 Instrucciones

6 Advanced Vector Extensions

- Preliminar
- Detección de las facilidades AVX
- Soporte a Half Precision
- Soporte para fusión de productos y sumas

# Detección de soporte a Half Precision

Las instrucciones de conversión entre half y single precision, **VCVTPH2PS** y **VCVTPS2PH**, requiere comprobar estar soportada por el procesador. El procesador siempre opera en single precision pero luego se puede convertir el resultado.

```
supports_VPCLMULQDQ:  
    mov eax, 1  
    cpuid  
    and ecx, 038000000H  
    cmp ecx, 038000000H; checkea flags OSXSAVE AVX y F16C.  
    jne not_supported  
; El Procesador soporta AVX, F16C, y el OS habilitó XGETBV.  
    mov ecx, 0; ecx=0 para leer registro XCRO  
    XGETBV ; EDX:EAX = XCRO  
    and eax, 06H  
    cmp eax, 06H; habilitó el OS el soporte de estado para XMM y YMM?  
    jne not_supported  
    mov eax, 1  
    jmp done  
not_supported:  
    mov eax, 0  
done:  
    ret
```

1 Fundamentos

2 Procesamiento de Señales digitales

3 Modelo de ejecución SIMD

4 Implementaciones SIMD en x86

5 Instrucciones

6 Advanced Vector Extensions

- Preliminar
- Detección de las facilidades AVX
- Soporte a Half Precision
- Soporte para fusión de productos y sumas

# Detección de soporte a Half Precision

# Detección de soporte a Half Precision

- Mejoran las capacidades aritméticas de alto rendimiento.

## Detección de soporte a Half Precision

- Mejoran las capacidades aritméticas de alto rendimiento.
- Abarcan multiplicación-suma fusionada, multiplicación-resta fusionada, multiplicación fusionada con suma/resta intercalada, multiplicación con signo invertido en multiplicación-suma fusionada y multiplicación-resta.

## Detección de soporte a Half Precision

- Mejoran las capacidades aritméticas de alto rendimiento.
- Abarcan multiplicación-suma fusionada, multiplicación-resta fusionada, multiplicación fusionada con suma/resta intercalada, multiplicación con signo invertido en multiplicación-suma fusionada y multiplicación-resta.
- Las extensiones **FMA** agregan 36 instrucciones de punto flotante de 256 bit para realizar cálculos en vectores de 256 bit e instrucciones **FMA** escalares y de 128 bit adicionales.

## Detección de soporte a Half Precision

- Mejoran las capacidades aritméticas de alto rendimiento.
- Abarcan multiplicación-suma fusionada, multiplicación-resta fusionada, multiplicación fusionada con suma/resta intercalada, multiplicación con signo invertido en multiplicación-suma fusionada y multiplicación-resta.
- Las extensiones **FMA** agregan 36 instrucciones de punto flotante de 256 bit para realizar cálculos en vectores de 256 bit e instrucciones **FMA** escalares y de 128 bit adicionales.
- Las extensiones **FMA** también incorporan 60 instrucciones de punto flotante de 128 bit para procesar datos vectoriales y escalares de 128 bit.

## Detección de soporte a Half Precision

- Mejoran las capacidades aritméticas de alto rendimiento.
- Abarcan multiplicación-suma fusionada, multiplicación-resta fusionada, multiplicación fusionada con suma/resta intercalada, multiplicación con signo invertido en multiplicación-suma fusionada y multiplicación-resta.
- Las extensiones **FMA** agregan 36 instrucciones de punto flotante de 256 bit para realizar cálculos en vectores de 256 bit e instrucciones **FMA** escalares y de 128 bit adicionales.
- Las extensiones **FMA** también incorporan 60 instrucciones de punto flotante de 128 bit para procesar datos vectoriales y escalares de 128 bit.
- Éstas últimas operaciones aritméticas cubren multiplicación-suma fusionada, multiplicación-resta fusionada, multiplicación invertida con signo tanto sobre multiplicación-suma fusionada, como multiplicación-resta fusionada.