# Práctica 3 - Modelo de procesamiento SIMD

Organización del Computador 2

2do Cuatrimestre 2016

La práctica se divide en secciones, para cada una se sugiere un conjunto de instrucciones útiles para resolver los ejercicios.

## 1. Instr. de movimiento de datos y aritméticas básicas

- Mov. de datos: **movd**, **modq**, **movdqu**
- Aritméticas: **paddb**, **paddw**, **paddd**, **paddq**, **psubb**, **psubw**, **psubd**, **psubq**

**Ejercicio 1**

Escriba las siguientes funciones en lenguaje ensamblador:

a) *void SumarVectores(char *vectorA, char *vectorB, char *vectorResultado, int dimension)*

b) *void RestarVectores(char *vectorA, char *vectorB, char *vectorResultado, int dimension)*

a) ¿Qué sucede si la suma de dos componentes de los vectores supera el valor 255? ¿Qué diferencia hay entre las instrucciones **paddusw** y **paddsw**? ¿Y entre **paddusb** y **paddsb**?

b) ¿Qué cambios deberían realizarse sobre las funciones anteriores si el tipo de datos es: a) *short*, b) *int* y c) *long long int*?

**Nota**: Puede asumir que la dimensión de los vectores es de un tamaño múltiplo de la cantidad de elementos que procesa simultáneamente.

## 2. Instr. de comparación, lógicas y de nivel de bit

- Comparación: **pcmpgtb**, **pcmpgtw**, **pcmpgtd**, **pcmpeqb**, **pcmpeqw**, **pcmpeqd**
- Lógicas: **pand**, **por**, **pxor**, **pandn**
- Nivel de Bit: **psrlw**, **psrld psrlq**, **psrldq**, **pslld**,**psllq**,**pslldq**

**Ejercicio 2**

Escriba las siguientes funciones en lenguaje ensamblador:

a) *void InicializarVectorEnCero(char *vectorA, int dimension)*

b) *void InicializarVector(short *vectorA, short valorInicial, int dimension)*

c) *void MultiplicarVectorPorPotenciaDeDos(int *vectorA, int potencia, int dimension)*

d) *void DividirVectorPorPotenciaDeDos(int *vectorA, int potencia, int dimension)*

e) *void FiltrarMayoresA(short *vectorA, short umbral, int dimension)*
Pone en **unos** (*0xF..F*) aquellos elementos del vector cuyo valor es mayor a *umbral* y en **ceros** (*0x0..0*) aquellos elementos que son **menores iguales**.

a) Qué cambios debería hacerles a las funciones anteriores en caso de que la dimensión de los vectores no sea múltiplo de la cantidad de elementos que procesa simultáneamente?

# 3. Instr. de movimiento de datos y aritméticas complejas

- Desempaquetado: **punpcklbw**, **punpcklwd**, **punpckldq**, **punpckhbw**, **punpckhwd**, **punpckhddq**
- Aritméticas: **pmullw**, **pmulld**, **pmulhw**, **pmulhd**, **pmaddwd**, **pmaxub**, **pmaxuw**, **pmaxud**, **pminub**, **pminuw**, **pminud**
- Empaquetado: **packsswb packssdw**, **packuswb packusdw**

**Ejercicio 3**

Escriba las siguientes funciones en lenguaje ensamblador:

a) *void ExtenderTamañoVector(unsigned char *vector, unsigned short *vectorExtendido, int dimension)*
Para cada elemento de *vector*, cuyo tamaño es de **1 Byte**, lo extiende a **1 Word** preservando su signo **positivo**.

b) *void MultiplicarVectores(short *vectorA, short *vectorB, int *vectorResultado, int dimension)*

c) *int ProductoInterno(short *vectorA, short *vectorB, int dimension)*

d) *void Maximos(char *vectorA, char *vectorB, char *vectorResultado, int dimension)*

e) *void SepararMaximosYMinimos(char *vectorA, char *vectorB, int dimension)*
Deja en *vectorA* los máximos y en *vectorB* los mínimos. Es decir, para cada $i$, *vectorA[i]* $= max($*vectorA[i]*,*vectorB[i]*$)$ y *vectorB[i]* $= min($*vectorA[i]*,*vectorB[i]*$)$

f) *void SumarYRestarVectores(int *vectorA, int *vectorB, int* vectorResultado, int dimension)*
Es decir, el *vectorResultado* tiene que seguir el siguiente patrón:

$$vectorResultado = (a_1 + b_1, a_2 - b_2, a_3 + b_3, a_4 - b_4, ...)$$

a) Qué cambios habría que hacerle a la función **ExtenderTamañoVector** para que pueda extender elementos con signos?

b) Qué cambios habría que hacerle a la función **MultiplicarVectores** si el tipo de datos de los elementos de *vectorResultado* fuese *short*?

c) Qué diferencia hay entre las instrucciones **packuswb** y **packsswb**?

# 4. Instr. de reordenamiento de datos

- reordenamiento: **pshufb**, **pshufw**, **pshufd**

**Ejercicio 4**

Escriba las siguientes funciones en lenguaje ensamblador:

a) *void Intercalar(char *vectorA, char *vectorB, char *vectorResultado, int dimension)*

a) Cómo haría función *Intercalar* si no dispone de las instrucciones de reordenamiento? Dé 2 maneras alternativas.

# 5. Instr. de punto flotante

- Mov. de datos: **movups**, **movaps**, **movupd**, **movapd**
- Aritméticas: **addps**, **addpd**, **subps**, **subpd**, **mulps**, **mulpd**, **divps**, **divpd**, **sqrtps**, **sqrtpd**

**Ejercicio 5**

Escriba las siguientes funciones en lenguaje ensamblador:

a) *void SumarVector(float *vectorA, float *vectorB, float *vectorResultado, int dimension)*

b) *void RestarVector(float *vectorA, float *vectorB, float *vectorResultado, int dimension)*

c) *void MultiplicarVector(float *vectorA, float *vectorB, float *vectorResultado, int dimension)*

d) *void DividirVector(float *vectorA, float *vectorB, float *vectorResultado, int dimension)*

e) *void NormalizarVector(float *vectorA, float *vectorResultado, int dimension)*

a) Qué cambios debería hacerles a las funciones anteriores si el tipo de dato ahora es punto flotante de **doble precisión**?

b) Qué diferencia hay entre la instrucción **addps** y **addss**? Y entre **addpd** y **addsd**?

# 6. Instr. de conversión de tipos de datos

- conversión: **cvtdq2ps**, **cvtps2dq**, **cvtdq2pd**, **cvtpd2dq**

**Ejercicio 6**

Escriba las siguientes funciones en lenguaje ensamblador:

a) *void ProductoEscalar(short *vectorA, float escalar, int dimension)*

b) *void ParteEntera(float *vectorA, int *vectorResultado, int dimension)*

c) *void Normalizar(int *vectorA, float *vectorNormalizado, int dimension)*

a) Qué cambios debería hacerles a las funciones anteriores si el tipo de dato ahora es punto flotante de **doble precisión**?

# 7. Ejercicios sobre imágenes

Una imagen está representada por una matriz de *alto* × *ancho*, donde cada posición representa un punto de color de la misma. A este punto se lo denomina *pixel*. En una imagen en escala de grises (de 8 bits), los colores varían desde el 0 (color negro) al 255 (color blanco) y cada *pixel*, desde luego, ocupa **1 byte**.

A su vez, en una imagen a color (de 24 bits), cada pixel está formado por 3 componentes (o canales): el rojo (**R**), el verde (**G**) y el azul (**B**). Cada una de estas componentes ocupa **1 Byte** (haciendo que el tamaño total del pixel sea de **3 Bytes**) y las distintas combinaciones posibles de las 3 generan los diferentes colores (la disposición en memoria de estás componentes es: primera la azul, luego la verde y por última la roja).

**Ejercicio 7**

Se desea realizar una función que dada una imagen en escala de grises invierta los colores, es decir, el realice el cálculo $255 - p$ para todos los pixeles $p$ de la imagen de origen.

El prototipo de la función es:

*void Invertir(unsigned char *imagenFuente, unsigned char *imagenDestino, int alto, int ancho)*

**Ejercicio 8**

Se desea realizar una función que dada una imagen en color retorne genere la misma imagen pero en escala de grises. Para hacer esto, se emplea la siguiente función:

$$imagenDestino(p) = max(R, G, B) \text{ para todo pixel } p \text{ de la } imagenFuente$$

El prototipo de la función es:

*void MonocromatizarInfinito(unsigned char *imagenFuente, unsigned char *imagenDestino, int alto, int ancho)*

**Ejercicio 9**

Se desea implementar la función combinar que dadas 2 imágenes de igual tamaño y en escala de grises retorna una tercera formada a partir de estas 2. Cada pixel de la imagen generada se forma de la siguiente manera:

$$imagenDestino(i, j) = \frac{alpha \cdot (imagenFuenteA(i, j) - imagenFuenteB(i, j))}{255} + imgB(i, j)$$

El prototipo de la función es:

*void Combinar (unsigned char* imagenFuenteA, unsigned char* imagenFuenteB, unsigned char* imagenDestino, int ancho, int alto, float alpha)*

**Ejercicio 10**

Se desea realizar una función que dada una imagen en color retorne 3 imágenes en escala de grises, donde la primera imagen está formada por las componentes rojas (R) de la imagen de entrada, la segunda por las componentes verdes (G) y la última por las componentes azules (B).

El prototipo de la función es:

*void SepararComponentes(unsigned char *imagenFuente, unsigned char *imagenDestinoR, unsigned char *imagenDestinoG, unsigned char *imagenDestinoB, int alto, int ancho)*

**Ejercicio 11**

Implementar la función umbralizar que genera una imagen de tres colores, blanco, gris y negro determinada por la imagen fuente respetando la siguiente función:

$$imagenDestino(p) = \begin{cases} 0 & p \leq umbral\_minimo \\ 128 & umbral\_minimo < p \leq umbral\_maximo \\ 255 & p > umbral\_maximo \end{cases}$$

para todo pixel $p$ de $imagenFuente$

El prototipo de la función es:

*void Umbralizar (unsigned char* imagenFuente, unsigned char* imagenDestino, int ancho, int alto)*

**Ejercicio 12**

Implementar la función de erosión que le aplica la siguiente operación a la imagen (en escala de grises) de entrada:

$$imgD(i,j) = min( \begin{array}{ccccc} imgF(i-1,j-1) & , & imgF(i-1,j) & , & imgF(i-1,j+1) \\ imgF(i+1,j-1) & , & imgF(i+1,j) & , & imgF(i-1,j+1) \\ imgF(i,j=1) & , & imgF(i,j) & , & imgF(i,j+1)) \end{array}$$

Donde $imgF$ es la imagen fuente y $imgD$ el destino. Los indices $i$ y $j$ corresponden a las coordenadas en la imagen.

El prototipo de la función es:

*void Erosion(unsigned char imagenFuente, unsigned char *imagenDestino, int alto, int ancho)*

Determinar el rango en el cual se puede realizar esta cuenta, para evitar indefiniciones. Fuera del rango, la matriz resultante no debe ser modificada.

**Ejercicio 13**

Implementar la función de suavizado que le aplica la siguiente operación a la imagen (en escala de grises) de entrada:

$$\begin{array}{rccccccc} imgD(i,j) = & imgF(i-1,j-1)\cdot 1/16 & + & imgF(i-1,j)\cdot 2/16 & + & imgF(i-1,j+1)\cdot 1/16 & + \\ & imgF(i+0,j-1)\cdot 2/16 & + & imgF(i+0,j)\cdot 4/16 & + & imgF(i+0,j+1)\cdot 2/16 & + \\ & imgF(i+1,j-1)\cdot 1/16 & + & imgF(i+1,j)\cdot 2/16 & + & imgF(i+1,j+1)\cdot 1/16 & \end{array}$$

Donde $imgF$ es la imagen fuente y $imgD$ el destino. Los indices $i$ y $j$ corresponden a las coordenadas en la imagen.

El prototipo de la función es:

*void Suavizar(unsigned char imagenFuente, unsigned char *imagenDestino, int alto, int ancho)*

Determinar el rango en el cual se puede realizar esta cuenta, para evitar indefiniciones. Fuera del rango, la matriz resultante no debe ser modificada.

**Nota**: Para generar el pixel $imagenDestino(i,j)$ en la imagen destino, se debe operar sobre enteros, por lo que la aplicación de la función descripta debe ser adaptada para tal caso.

**Ejercicio 14**

Implementar la función promediar vecinos que dada una imagen (en escala de grises) de entrada, le aplica la siguiente fórmula a todos los píxeles:

$$imgD(i,j) = \frac{imgF(i,j-1) + imgF(i,j+1) + imgF(i-1,j) + imgF(i+1,j)}{4}$$

Donde $imgF$ es la imagen fuente y $imgD$ el destino. Los indices $i$ y $j$ corresponden a las coordenadas en la imagen.

El prototipo de la función es:

*void PromediarVecinos(unsigned char* imagenFuente, unsigned char* imagenDestino, int alto, int ancho)*

Determinar el rango en el cual se puede realizar esta cuenta, para evitar indefiniciones. Fuera del rango, la matriz resultante no debe ser modificada.