MMX

```
MMX es un conjunto de instrucciones SIMD (Single Instruction Multiple Data) creadas por Intel. La tecnología MMX fue diseñada para mejorar de forma sustancial el rendimiento de las aplicaciones multimedia y de telecomunicaciones. Se basa en un nuevo juego de instrucciones, que se añadían a las ya existentes en la arquitectura 80×86, y en nuevos tipos de datos de 64 bits. Dichas instrucciones trabajaban en paralelo sobre múltiples elementos de datos empaquetados en cantidades de 64 bits.
```

SSE (Streaming SIMD Extensions)

```
Las SSE (SSE2, SSE3, SSSE3, SSE4 (4.1, 4.2, a) son unas extensiónes al grupo de instrucciones MMX, que también fueron desarrolladas por Intel, concretamente, para sus procesadores Pentium III (1999). Están especialmente pensadas para decodificación de MPEG2 (códec vinculado a los DVD), procesamiento de gráficos tridimensionales y software de reconocimiento de voz. Con la tecnología SSE, los microprocesadores x86 fueron dotados de setenta nuevas instrucciones y de ocho registros nuevos: del xmm0 al xmm7. Estos registros tienen una extensión de 128 bits (es decir que pueden almacenar hasta 16 bytes de información cada uno).
```

NET Bubble Sort SEE (comparando assembler + C)

- ¿Qué hace el programa?

  ```
    Ordena arrays con contenido aleatorio de valor double, usando el algoritmo BubbleSort implementado en C y en Inline ensamblador. Se ordenan arrays de longitud 2 cuya longitud  crece expotencialmente despues de cada iteracion hasta tener la longitud 66537, y muestra  el tiempo transcurrido en cada caso, comparando ambas implementaciones.
  ```

- ¿Se utilizan extensiones MMX o SSE?

  ```
    SSE, ya que se pueden reconocer instrucciones del set SSE2 tales como minpd, maxpd, movapd...
  ```

- Comenta el funcionamiento de la función: void sortDoublesSSE(Int32 byteCount, double* values)

  ```
    // TO DO
  ```

- ¿Qué ganancia obtenemos con el algoritmo MMX/SSE con respecto al algoritmo secuencial?

  ```
    A medida que la longitud del array va aumentando, podemos observar que la ganancia se va estabilizando en el 100%, siendo la implementacion en SSE el doble de rapida que en C.
  ```

- Realiza una batería de pruebas y muéstralo utilizando gráficas explicativas.

longitud	Tiempo cms	Tiempo ssems	Ganancia %
2	0	0	00,0
4	0	0	00,0
8	0	0	00,0
16	0	0	00,0
32	0	0	00,0
64	0	0	00,0
128	0	0	00,0
256	0	0	00,0
512	0	0	00,0
1024	1	1	00,0
2048	7	4	75,0
4096	34	21	61,9
8192	163	86	89,5
16384	662	358	84,9
32768	2741	1393	96,8
65536	11408	5742	98,7
			v
			(cms / ssems * 100 - 100)

NET Data Transfer (comparando assembler + C)

- ¿Qué hace el programa?

```
  Medir cuanto tarda el sistema en transferir datos (4 bytes (sizeof: int ) de un registro de memoria a otro una cantidad de veces (definido en ITERATION), implementado en C y en Inline ensamblador, mostrando la diferencia entre ambos. Además permite reejecutar dicha prueba para obtener unos datos concluyentes sobre este.
```

- Explica las siguientes instrucciones: shufpd, cmpltpd, movmskpd
  - shufpd: Baraja dos pares de valores de precision de coma flotante de mxx1 y mxx2, y devuelve el resultado a mxx1.
  - cmpltpd: Compara si dos valores de precision de coma flotante son "menores que" entre si.
  - movmskpd: Extrae la mascara de signo de un valor de precision de coma flotante en mmx.

- Explica el funcionamiento de la siguiente función: int DataTransferOptimised(int* piDst, int* piSrc, unsigned long SizeInBytes)

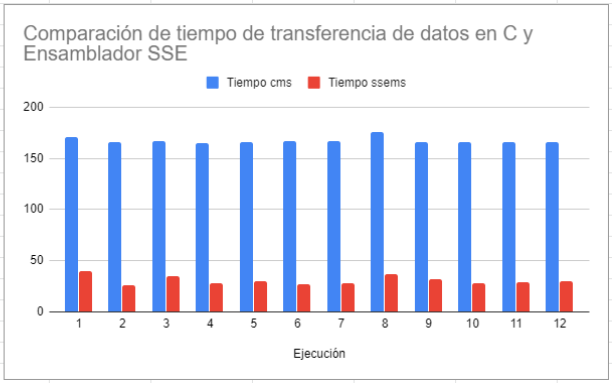
```
// TO DO
```

- ¿Qué ganancia obtenemos con el algoritmo optimizado mediante extensiones SIMD con respecto al algoritmo secuencial?

```
  Si comparamos los resultado tras varias iteraciones, podemos comprobar que la implementación en ensamblador llega a ser hasta 5 veces más rapida que la implementada en C.
```

- Realiza una batería de pruebas y muéstralo utilizando gráficas explicativas.

Ejecución	Tiempo cms	Tiempo ssems	Ganancia %
1	171	40	327,5
2	166	26	538,5
3	167	35	377,1
4	165	28	489,3
5	166	30	453,3
6	167	27	518,5
7	167	28	496,4
8	176	37	375,7
9	166	32	418,8
10	166	28	492,9
11	166	29	472,4
12	166	30	453,3



NET Inner Product (comparando assembler + C)

- ¿Qué hace el programa?

```
  Suma la multiplicación de arrays rellenos de valores aleatorios de distinto tipo implementado de diferentes maneras, en C, en ensamblador usando instrucciones de SSE2 y otra con instrucciones de SS3. Cabe considerar de que el programa además usa extensiones de Microsoft.
```

- Comenta la siguiente función: float sse3_inner(const float* a, const float* b, unsigned int size)

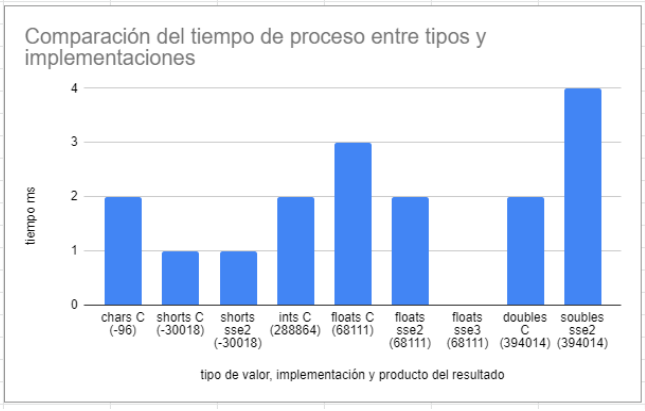
```
// TO DO
```

- ¿Qué ganancia obtenemos con el algoritmo optimizado mediante extensiones SIMD con respecto al algoritmo secuencial?

```
  Por lo que podemos ver segun los resultados, la ganancia esta vez depende del set que se usa y sobre que tipo de valor se usa, aún así, el resultado es el de siempre, favorable para la implementacion en ensamblador, los calculos realizados con floats parecen ser el triple de rapidos para el set sse3, y en el resto de tipos parece llegar a duplicar la velocidad de la implementacion en C.
```

- Realiza una batería de pruebas y muéstralo utilizando gráficas explicativas.

tiempo ms	tipo, implementación y producto del resultado	
2	chars C	(-96)
1	shorts C	(-30018)
1	shorts sse2	(-30018)
2	ints C	(288864)
3	floats C	(68111)
2	floats sse2	(68111)
0	floats sse3	(68111)
2	doubles C	(394014)
4	soubles sse2	(394014)



NET MatrixVectorMult (comparando assembler + C)

- ¿Que hace el programa?

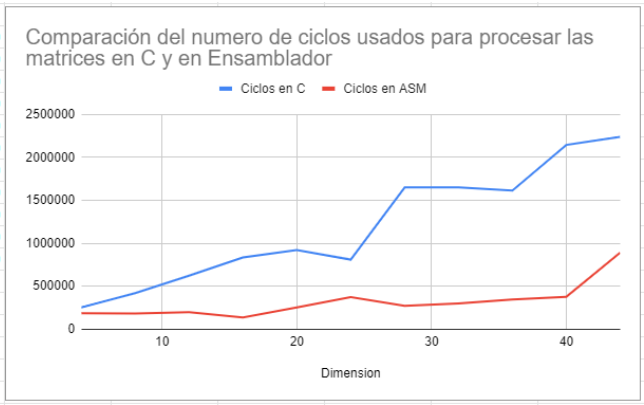
```
  Pide al usuario un tamaño, y general un vector y una matriz de dicho tamaño con
los valores de su propia posición empezando desde cero, despues realiza un
calculo el cual es la suma de la multiplicación itinerada de la matriz,
implementada en C y en ensamblador, y muestra la comparativa entre ambas,
moestrando cual ha usado menos ciclos de reloj y el tiempo ahorrado al usar
ensamblador.
```

- ¿Qué ganancia obtenemos con el algoritmo optimizado mediante extensiones SIMD con respecto al algoritmo secuencial?

```
  Despues de re-ejecutar el programa con varias dimensiones para el vector,
podemos presenciar que la ganancia se va estabilizando al rededor de un 75%,
siendo la funcion en ensamblador un 75%~ más rapida que la implementada en C.
```

- Realiza una batería de pruebas y muéstralo utilizando gráficas explicativas.

Dimension	Ciclos en C	Ciclos en ASM	Ganancia %
4	254656	187202	26,0
8	420612	184612	56,0
12	623062	198082	68,0
16	835010	138010	83,0
20	920800	254316	72,0
24	810756	375530	53,0
28	1650394	274562	83,0
32	1651580	300886	81,0
36	1613278	346586	78,0
40	2145108	377954	82,0
44	2238058	892204	60,0



Fuentes:

- https://books.google.es/books?id=bt3ZjeB4v9gC&printsec=frontcover&hl=es#v=onepage&q&f=false
- https://books.google.es/books?id=c3SlgrqMid4C&printsec=frontcover&hl=es#v=onepage&q&f=false
- https://books.google.es/books?id=C3_WIQOYE2EC&printsec=frontcover&hl=es#v=onepage&q&f=false
- https://books.google.es/books?id=FGxOp0lAljUC&printsec=frontcover&hl=es#v=onepage&q&f=false
- http://softpixel.com/~cwright/programming/simd/
- https://docs.oracle.com/cd/E18752_01/html/817-5477/eojdc.html
- https://docs.oracle.com/cd/E26502_01/html/E28388/eojde.html
- http://www.jegerlehner.ch/intel/IntelCodeTable.pdf
- https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf

Preguntas:

- Porque en data transfer se multiplica por size(int) y despues se divide?
- Porque en Inner product, la funcion "sse_inner_d" y "sse_inner" no está realmente implementada en ningun bloque __asm?