# ARQUITECTURA DE SISTEMAS DISTRIBUIDOS.
# 3º GIC. PRÁCTICA 4: DLP (VOLUNTARIA).
# RENDIMIENTO UTILIZANDO EL NÚCLEO VECTORIAL.

## 1. OBJETIVOS Y PREPARACIÓN.

En esta práctica vamos a estudiar la optimización de código mediante el uso de las extensiones multimedia o núcleos vectoriales disponibles en los procesadores P4.

La historia de las extensiones multimedia en la familia x86 ha sido resumidamente:

- Comienza con el Pentium MMX (1994), en el que se añadieron un conjunto de 8 registros de 64 bits (2x32 bits) y una serie de instrucciones que permitían el procesado SIMD (*Single Instruction Múltiple Data*) de datos enteros. Estas extensiones se denominaron MMX.
- El siguiente paso se da en el Pentium III, el cual contiene un conjunto de 8 registros de 128 bits (4x32 bits) y una serie de instrucciones para el procesado SIMD de flotantes en simple precisión. A estas instrucciones se les denomina Streamming SIMD Extensions (SSE).
- Posteriormente, en el P4 se incluyen instrucciones (SSE2 y SSE3) para poder emplear los registros SSE con datos enteros y flotantes de simple o doble precisión (4x32bits o 2x64 bits).
- Actualmente existen núcleos de 512 bits (AVX2)

Tanto los niveles de caché como su tamaño, número de vías y tamaño de la línea tendrán una influencia muy importante en el rendimiento de las pruebas que haremos en esta práctica.

**El alumno deberá:**

- entender **cómo está organizado el código que se proporciona** con este enunciado.
- Además, conocer **cómo se ubican en memoria cada una las variables** que aparecen en éste en base a sus declaraciones y entender el subconjunto de instrucciones multimedia IA32 que se proporcionan abajo.
- Hallar los tamaños de L1,L2,L3 de la máquina a probar, y si se puede el AB de la RAM. Por ejemplo, usando CPUZ (https://www.cpuid.com/softwares/cpu-z.html) .

A continuación se presentan las instrucciones IA32 que serán necesarias para la realización de la práctica, con su referencia a "*IA32 Intel Architecture Software Developer's Manual. Volume 2: Instruction Set Reference*". Nos restringiremos a:

- Todos los operandos en registro o memoria con los que trabajaremos serán de 32 bits (o de 128 para registros **xmm** de 128 bits y accesos multimedia) aunque algunas instrucciones admiten otros tamaños de operandos.
- Los operandos inmediatos pueden cambiar de tamaño, pero no afecta a los benchmark de esta práctica.
- Y los direccionamientos que usaremos serán (aunque algunas instrucciones admiten otros): **[r], [r+offset]**

**NOTACIÓN: r = registro ; m = dirección de memoria. imm = constante inmediata**

**ARITMÉTICAS ENTERAS:**

**add r/m, r/m/imm[32,16 o 8]** *(Add – pág 3.21)*

**lea r, m** *(Load Effective Address – pág 3.372)*

**sbb r/m, r/m/imm[32,16 o 8]** *(Integer SuBtraction with Borrow – pág 3.685)*

**sub r/m, r/m/imm[32,16 o 8]** *(Subtract – pág 3.729)*

**cmp r, r/m/imm[32,16 o 8]** *(CoMPare two operands – pág 3.82)*

**SALTOS:**

**jne etiqueta** *(Jump if Not Equal – pág 3.352)*

**ACCESOS A MEMORIA:**

**mov r/m, r/m/imm[32,16 o 8]** *(Move – pág 3.430). Notas: Ambos operandos deben ser del mismo tamaño. Ambos operandos no pueden referenciar a memoria.*

**movss xmm/m, xmm/m** *(Move Scalar Single-precision floating-point values – pág 3.483)*

**movaps xmm/m128, xmm/m128** *(Move Aligned Packed Single-precision floating-point values – pág 3.441*)

**ARITMÉTICAS MULTIMEDIA:**

**addps xmm, xmm/m** *(Add Packed Single-precision floating-point values – pág 3.25)*

**addss xmm, xmm/m** *(Add Scalar Single-precision floating-point values – pág 3.29)*

**mulps xmm,xmm/m128** *(Multiply Scalar Single-precision floating-point values – pág 3.496)*

**mulss xmm, xmm/m** *(Multiply Scalar Single-precision floating-point values – pág 3.500)*

**shufps xmm, xmm/m128, imm8** *(Shuffle Packed Single-precision floating-point values- pág 3.703)*

**sqrtps xmm, xmm/m128** *(Computate Square Roots of Packed Single-precision floating-points values – pág 3.713)*

Como registros de programación, podrá hacer uso de **eax, ebx, ecx, edx, esi** y **edi** de 32 bits y **xmm0, ... xmm7**.

NOTA: la instrucción **cmp** no se debe separar, en general, del salto **jne**, pues en los x86 hay un bit de estado que guarda el resultado de la última operación aritmética (de forma que si entre **cmp** y **jne** hubiese otra instrucción aritmética, tal bit podría ser modificado y perderse el resultado de la comparación).

## 2. REALIZACIÓN DE LA PRÁCTICA.

Se da un proyecto con dos configuraciones: Debug y Release. Abrir el proyecto que se proporciona y verificar que se encuentran activas las siguientes opciones de compilación en Propiedades de configuración → C/C++

Para Debug (sólo servirá por si se comete algún fallo a escribir código):

→ General: Formato de la información de depuración → Base datos de prog. /Zi,
→ Optimización → Optimización: Deshabilitado
→ Optimización → expansión de funciones insertadas: Deshabilitado
→ Generación de código → Comprobaciones básicas en tiempo de ejecución: Predeterminado
→ Generación de código → Sin instrucciones mejoradas (/arch:ia32)
→ Generación de código → Modelo de punto flotante: Rápido (/fp:fast)

Para Release (servirá para medir prestaciones, rendimiento, etc.):

→ General: Formato de la información de depuración → Base datos de prog. /Zi,
→ Optimización → Optimización: Optimización completa (/Ox)
→ Optimización → expansión de funciones insertadas: Deshabilitado
→ Optimización → Tamaño o velocidad: Favorecer código rápido (`Ot`).
→ Generación de código → Comprobaciones básicas en tiempo de ejecución: Predeterminado
→ Generación de código → Habilitar conjunto de instrucciones mejorado: SSE2 (/arch:SSE2)
→ Generación de código → Modelo de punto flotante: Rápido (/fp:fast)

a) Ejecutar en modo Release, utilice *Depurar → Iniciar sin depurar (o CTRL+F5*). Relacionar los cambios de ciclos por elemento para el bucle SAXPY en lenguaje C (ya se da hecho) con los tamaños de los cachés. Comprobar que los saltos en tiempos de ejecución se han producido al desbordar las cachés.
b) Poner un punto de ruptura y ver como el compilador genera instrucciones vectoriales. Ver también la función `init(…)` del proyecto.
c) Implementar el bucle SAXPY de otras dos formas: en ensamblador haciendo uso de las instrucciones multimedia que operan sobre un único dato (sufijo *ss*), y en ensamblador haciendo uso de las que operan sobre múltiples datos (sufijo *ps*); `codigo_sse_escalar` y `codigo_sse_vectorial`, respectivamente. Para insertar instrucciones en ensamblador dentro del código C (ensamblado en línea) puede utilizar como ejemplo lo visto en b).
d) Ejecute el programa con *Depurar → Iniciar sin depurar (o CTRL+F5*).
    1. Si ha implementado de forma incorrecta las funciones solicitadas anteriormente aparecerá el mensaje *WRONG RESULTS: check your code* durante la ejecución. Corríjalo, cambiando al modo Debug, y la tecla F5.
    2. Una vez comprobadas las implementaciones rellene la tabla de resultados.

e) Otras pruebas.
    1. Desenrollar en ensamblador dos iteraciones para el código SSE vectorial (se procesarían 8 elementos por iteración) y para el código SSE escalar, entrelazando y reordenando. Anotar la aceleración conseguida para diferentes tamaños
    2. Probar con diferentes zancadas de recorrido de los vectores (en lugar de i++ en el bucle *for*, usar i+=2, i+=4, etc.). De esa forma el *Miss rate* cambia, y deben notarse diferencias mayores cuando los vectores no caben en los diferentes niveles de cachés
    3. Realizar las diferentes pruebas con otro procesador más moderno, usando núcleos vectoriales AVX o AVX2

# ASD. 3º GIC. PRÁCTICA 4: DLP (VOLUNTARIA).
## RENDIMIENTO USANDO EL NÚCLEO VECTORIAL.

| **PRUEBA 1**: **Comparación de las implementaciones.** | | | |
|---|---|---|---|
| | **C** | **Un único dato (ss)** | **Múltiples datos (ps)** |
| **Tiempo en ciclos para 512 elementos** | | | |
| **Ciclos por Elemento procesado para 512 elementos** | | | |
| **Aceleración respecto del código ensamblador escalar** | | **1.0** | |
| **CPI (y fórmula usada)** | | | |
| Justificar cualitativa y brevemente la diferencia de tiempos. | | | |
| ¿Qué diferencia habría entre el código que usa el núcleo vectorial (instrucciones sobre múltiples datos) y el código surgido del desenrollado de 4 iteraciones usando instrucciones escalares? | | | |

**Probar con diferentes número de elementos para ver como se degrada el rendimiento**

| **Tamaño que usan ambos vect. juntos** | **Núm. elem de x[]** | **C** | | **escalar** | | **vectorial** | |
|---|---|---|---|---|---|---|---|
| | | **cic/elem** | **Acel.** | **cic/elem.** | **Acel.** | **cic/elem.** | **aceleración** |
| | | | | | **1.0** | | |
| | | | | | **1.0** | | |
| | | | | | **1.0** | | |
| | | | | | **1.0** | | |
| | | | | | **1.0** | | |
| | | | | | **1.0** | | |
| | | | | | **1.0** | | |
| | | | | | **1.0** | | |
| | | | | | **1.0** | | |