*ARQUITECTURAS MULTIMEDIA*

3º Ingeniero Técnico en Informática de Sistemas

PRÁCTICA 1: PROGRAMACIÓN CON EXTENSIONES MULTIMEDIA
Marzo de 2010

# ÍNDICE

# ÍNDICE

# ÍNDICE

# ÍNDICE

# ÍNDICE

# 1. OBJETIVOS DE LA PRÁCTICA

- Aprender a utilizar los compiladores de ensamblador para arquitecturas Intel bajo el sistema operativo Linux (compiladores "NASM" y "GAS").

- Saber integrar código ensamblador dentro de archivos fuente escritos en C o C++ (compiladores "GCC" e "ICC").

- Conocer las instrucciones orientadas al procesamiento multimedia que Intel ha ido agregando a su arquitectura en sucesivas entregas (MMX, SSE, SSE2, SSE3, SSE4...).

- Valorar y comparar las mejoras que supone la paralelización de operaciones aritméticas (tanto en punto fijo como en punto flotante) presentes en algoritmos de procesamiento multimedia, frente a la implementación de esos mismos algoritmos con técnicas estrictamente secuenciales.

- Ser consciente de la importancia que tiene diseñar nuestros algoritmos teniendo presente cómo es la arquitectura de la máquina sobre la que éstos se van a ejecutar.

# 2. HARDWARE Y SOFTWARE NECESARIO

## 2.1 - HARDWARE NECESARIO

- Máquinas con CPU's Intel Pentium 4 o superiores (Intel Core Duo, Intel Core 2 Duo, Intel Quad Core, etc).

## 2.2 - SOFTWARE NECESARIO

- Sistema Operativo: Linux (distribuciones probadas en el laboratorio: Ubuntu y Fedora.

- Compiladores ICC (/opt/intel/Compiler/<versión>/bin/ia32/icc -para código C- y /opt/intel/Compiler/<versión>/bin/ia32/icpc -para código C++). Son gratuitos; pueden descargarse desde http://software.intel.com/en-us/articles/non-commercial-software-download/#compilers.

## 2. HARDWARE Y SOFTWARE NECESARIO

### SOFTWARE NECESARIO (continuación)

- Compiladores de ensamblador Intel bajo Linux:
  - GAS (GNU Assembler): Es "/usr/bin/as" y, al igual que GCC, suele formar parte de cualquier distribución de Linux.
  - NASM (Netwide Assember): Es "/usr/bin/nasm". Si no está preinstalado **(1)**, instalar el correspondiente paquete **(2)**.

**(1)**

```
[root@localhost ~]$ rpm -q nasm
nasm-0.98.39-5.fc7
```

**(2)**

```
[root@localhost ~]$ yum install nasm
```

# 3. PRIMERA TOMA DE CONTACTO CON NASM

## 3.1 - CARACTERÍSTICAS GENERALES DE NASM

- Utiliza una sintaxis similar a la del "Macro Assembler" (MASM) del DOS.

- Estructura del código fuente de un programa: Sección ".data" (constantes), sección ".bss" (variables) y sección ".text" (código).

- Operaciones de E/S: Realizables a través de llamadas al sistema operativo, con la interrupción software 80h (equivalente a la 21h del DOS). *En esta práctica no será necesario su uso* (si hay operaciones de E/S, éstas se implementarán en código C y no en ensamblador, que quedará reservado sólo para el uso de las instrucciones que formen parte de las extensiones multimedia que sea preciso utilizar).

- Edición del código fuente: Con el "vi" o con cualquier otro editor de texto.

# 3. PRIMERA TOMA DE CONTACTO CON NASM

## CARACTERÍSTICAS GENERALES DE NASM (continuación)

- Archivos intervinientes: <Fuente>.asm, <Objeto>.o y <Ejecutable>
- Compilación y lincaje:
  - Compilación (generación del fichero objeto):

    ```
    nasm -f elf <nombre>.asm
    ```

  - Lincaje (generación del fichero ejecutable):

    ```
    ld -o <nombre> <nombre>.o
    ```

- Más información: Archivo "asm-linux.pdf", en http://webs.um.es/einiesta/miwiki/doku.php?id=docencia

# 3. PRIMERA TOMA DE CONTACTO CON NASM

## 3.2 - PROGRAMA DE EJEMPLO EN NASM

- Código fuente ("hola.asm"):

```
section .data
    mensaje  db "hola mundo",0xA
    longitud equ $ - mensaje

section .text
    global _start    ; definimos el punto de entrada.
_start:
    mov edx,longitud ; EDX=longitud de la cadena.
    mov ecx,mensaje  ; ECX=cadena a imprimir.
    mov ebx,1        ; EBX=manejador del fichero (STDOUT)
    mov eax,4        ; EAX=función sys_write() del kernel.
    int 0x80         ; interrupción software 80h (llamada al kernel).
    mov ebx,0        ; EBX=código de salida al SO ($?).
    mov eax,1        ; EAX=función sys_exit() del kernel.
    int 0x80         ; interrupción software 80h (llamada al kernel).
```

# 3. PRIMERA TOMA DE CONTACTO CON NASM

## PROGRAMA DE EJEMPLO EN NASM (continuación)

- Compilación, lincaje y ejecución:

```
[einiesta@localhost holamundo]$ ls -l
total 16
-rw-rw-r-- 1 einiesta einiesta 529 mar  4 20:47 hola.asm
[einiesta@localhost holamundo]$ nasm -f elf hola.asm
[einiesta@localhost holamundo]$ ls -l
total 24
-rw-rw-r-- 1 einiesta einiesta 529 mar  4 20:47 hola.asm
-rw-rw-r-- 1 einiesta einiesta 736 mar  8 20:38 hola.o
[einiesta@localhost holamundo]$ ld -o hola hola.o
[einiesta@localhost holamundo]$ ls -l
total 32
-rwxrwxr-x 1 einiesta einiesta 440 mar  8 20:38 hola
-rw-rw-r-- 1 einiesta einiesta 529 mar  4 20:47 hola.asm
-rw-rw-r-- 1 einiesta einiesta 736 mar  8 20:38 hola.o
[einiesta@localhost holamundo]$ ./hola
hola mundo
```

# 4. PRIMERA TOMA DE CONTACTO CON GAS

## 4.1 - CARACTERÍSTICAS GENERALES DE GAS

- Utiliza una sintaxis propia, diferente de la utilizada por NASM (linux) y MASM (dos).
- Es el ensamblador estándar de Linux, y el que utilizaremos en las prácticas cuando incluyamos llamadas a extensiones multimedia en nuestro código C o C++.
- Estructura del código fuente de un programa: Similar a NASM, es decir, con sección ".data" (constantes), sección ".bss" (variables) y sección ".text" (código).
- Operaciones de E/S: Como en el caso de NASM, realizables a través de llamadas al sistema, con la interrupción software 80h.
- Edición del código fuente: Con el "vi" o con cualquier otro editor de texto.

# 4. PRIMERA TOMA DE CONTACTO CON GAS

## CARACTERÍSTICAS GENERALES DE GAS (continuación)

- Archivos intervinientes: <Fuente>.s, <Objeto>.o y <Ejecutable>
- Compilación y lincaje:
  - Compilación (generación del fichero objeto):

    ```
    as -o <nombre>.o <nombre>.s
    ```

  - Lincaje (generación del fichero ejecutable):

    ```
    ld -o <nombre> <nombre>.o
    ```

- Más información: Archivo "asm-linux.pdf", en http://webs.um.es/einiesta/miwiki/doku.php?id=docencia

# 4. PRIMERA TOMA DE CONTACTO CON GAS

## 4.2 - PRINCIPALES DIFERENCIAS ENTRE LA SINTAXIS GAS Y LA SINTAXIS NASM

- A los nombres de los registros se les añade el prefijo "%" (ejemplos: `%eax, %ebx, %ecx, %edx, %edi, %esi`, etc).

- El operando destino se coloca a la derecha, y el operando fuente a la izquierda (ejemplo: lo que en NASM es un "`mov ebx, eax`", en GAS es un "`movl %eax, %ebx`").

- El tamaño del resultado se establece explícitamente en las instrucciones "mov", utilizando los sufijos "b", "w" o "l" para, respectivamente, los tamaños "byte", "word" o "long word" (ejemplos: "`movw %bx,%ax`", "`movl (%ebx),%eax`").

- Los valores inmediatos han de ir precedidos por el carácter "$" (ejemplo: "`movl $0xf02, %ebx`).

- Otros matices ("`[.section]`" por "`section`", "`.globl`" por "`global`", "`ascii`" por "`db`", etc).

# 4. PRIMERA TOMA DE CONTACTO CON GAS

## 4.3 - PROGRAMA DE EJEMPLO EN GAS

- Código fuente ("hola.s"):

```
.section .data
    mensaje: .ascii "hola mundo \n"
    longitud = . - mensaje

.section .text
    .globl _start
_start: movl $longitud,%edx   # EDX=longitud de la cadena.
    movl $mensaje,%ecx        # ECX=cadena a imprimir.
    movl $1,%ebx              # EBX=manejador del fichero (STDOUT).
    movl $4,%eax              # EAX=función sys_write() del kernel.
    int $0x80                 # Interrupción software 80h (llamada al kernel).
    movl $0,%ebx              # EBX=código de salida al SO ($?).
    movl $1,%eax              # EAX=función sys_exit() del kernel.
    int $0x80                 # Interrupción software 80h (llamada al kernel).
```

# 4. PRIMERA TOMA DE CONTACTO CON GAS

## PROGRAMA DE EJEMPLO EN GAS (continuación)

- Compilación, lincaje y ejecución:

```
[einiesta@localhost holamundo]$ ls -l
total 8
-rw-rw-r-- 1 einiesta einiesta 522 mar  8 22:17 hola.s
[einiesta@localhost holamundo]$ as -o hola.o hola.s
[einiesta@localhost holamundo]$ ls -l
total 16
-rw-rw-r-- 1 einiesta einiesta 616 mar  8 22:18 hola.o
-rw-rw-r-- 1 einiesta einiesta 522 mar  8 22:17 hola.s
[einiesta@localhost holamundo]$ ld -o hola hola.o
[einiesta@localhost holamundo]$ ls -l
total 24
-rwxrwxr-x 1 einiesta einiesta 637 mar  8 22:18 hola
-rw-rw-r-- 1 einiesta einiesta 616 mar  8 22:18 hola.o
-rw-rw-r-- 1 einiesta einiesta 522 mar  8 22:17 hola.s
[einiesta@localhost holamundo]$ ./hola
hola mundo
```

# 5. PRIMERA TOMA DE CONTACTO CON ICC

## 5.1 - CARACTERÍSTICAS GENERALES DE ICC

- ICC, o "Intel C++", es un grupo de dos compiladores creados por Intel, el "icc" y el "icpc", que permiten, respectivamente, la implementación de aplicaciones utilizando los lenguajes C y C++.

- Es de libre distribución, y dispone de versiones para Windows y Linux, y también para diversas variantes de la arquitectura Intel (CPU's de 32 y 64 bits).

- La sintaxis de los dos compiladores (línea de comandos) es muy similar a la de los respectivos compiladores homólogos de GCC ("gcc" y "g++").

- Permiten la integración de código de GAS ("ensamblador en línea").

- Posibilitan el uso de extensiones multimedia (mediante ensamblador GAS en línea, mediante librerías en C o a través de clases de C++).

# 5. PRIMERA TOMA DE CONTACTO CON ICC

## 5.2 - INSTALACIÓN Y PUESTA EN MARCHA

- La descarga de la instalación se lleva a cabo desde http://software.intel.com/en-us/articles/non-commercial-software-download/#compilers. Hay que tener especial cuidado en elegir la versión apropiada del ICC. A fecha de Marzo de 2010, esta versión sería la siguiente: "*Intel C++ Compiler Professional Edition 11.1 for Linux*"; en cuanto a la arquitectura, si no se está seguro, elegir la versión de 32 bits, que es compatible con todo tipo de máquinas (aunque del Intel Core 2 Duo en adelante se podría elegir también la versión de 64 bits).

- Para instalar ICC, síganse las instrucciones correspondientes que se incluyen en el paquete de instalación. Resultado de la instalación (respetando las opciones propuestas por el asistente de instalación):

  - Directorio base de instalación: `/opt/intel`.
  - Ubicación de los compiladores: `/opt/intel/Compiler/<versión>/bin/ia32/icc` (compilador de C), y `/opt/intel/Compiler/<versión>/bin/ia32/icpc` (compilador de C++).
  - Ubicación del depurador: `/opt/intel/Compiler/<versión>/bin/ia32/idb`

# 5. PRIMERA TOMA DE CONTACTO CON ICC

## INSTALACIÓN Y PUESTA EN MARCHA (continuación)

- Una vez terminada la instalación, hay que proceder a realizar una "postinstalación":
  - Asegurarse de que al inicio de cada sesión se ejecuta el siguiente script. En caso necesario, actualizar ".bashrc" **(3)**.

    `/opt/intel/Compiler/<version>/bin/ia32/iccvars_ia32.sh`

  - Comprobar que la variable de entorno "LD_LIBRARY_PATH" está instanciada con la ruta de las librerías que utiliza ICC (`/opt/intel/Compiler/<versión>/lib/ia32`).

**(3)**

```
source /opt/intel/Compiler/<version>/bin/ia32/iccvars_ia32.sh
```

# 5. PRIMERA TOMA DE CONTACTO CON ICC

## 5.3 - COMPILACIÓN CON ICC

- Archivos intervinientes: <Fuente>.c y <Ejecutable>.
- Compilación: `icc <nombre>.c -o nombre`

**IMPORTANTE**:

Si algún fichero fuente incluye extensiones multimedia, puede ser necesario (aunque no siempre) obligar al compilador a que adapte el código máquina generado a la arquitectura Intel concreta en la que dicho código va a ejecutarse. Para ello, hay que utilizar el argumento "-ax" seguido de una letra que identifica la arquitectura. Por ejemplo, para compilar un programa en C con extensiones multimedia que vaya a ejecutarse sobre un Intel Core 2 Duo, habría que escribir:

```
icc -axT <nombre>.c -o nombre.
```

# 5. PRIMERA TOMA DE CONTACTO CON ICC

## COMPILACIÓN CON ICC (continuación)

Valores posibles de "-ax<Arquitectura>":

- -axT: Intel Core 2 Duo, Intel Core 2 Quad.
- -axP: Intel Core Duo.
- -axW: Intel Pentium IV.
- -axK: Intel Pentium III.

- Más información:
  - Archivo "resumen_icc.pdf", en http://webs.um.es/einiesta/miwiki/doku.php?id=docencia
  - Páginas del manual: `man icc`, o `konqueror man:icc`.

# 5. PRIMERA TOMA DE CONTACTO CON ICC

## 5.4 - COMPILACIÓN CON ICPC

- Archivos intervinientes: <Fuente>.cpp y <Ejecutable>.
- Compilación: `icpc <nombre>.cpp -o nombre`

**IMPORTANTE**:

Si algún fichero fuente incluye extensiones multimedia, puede ser necesario (aunque no siempre) obligar al compilador a que adapte el código máquina generado a la arquitectura Intel concreta en la que dicho código va a ejecutarse. Para ello, hay que utilizar el argumento "-ax" seguido de una letra que identifica la arquitectura. Por ejemplo, para compilar un programa en C++ con extensiones multimedia que vaya a ejecutarse sobre un Intel Core 2 Duo, habría que escribir:

```
icpc -axT <nombre>.c -o nombre.
```

# 5. PRIMERA TOMA DE CONTACTO CON ICC

## COMPILACIÓN CON ICPC (continuación)

Valores posibles de "-ax<Arquitectura>":

- -axT: Intel Core 2 Duo, Intel Core 2 Quad.
- -axP: Intel Core Duo.
- -axW: Intel Pentium IV.
- -axK: Intel Pentium III.

- Más información:
  - Archivo "resumen_icpc.pdf", en http://webs.um.es/einiesta/miwiki/doku.php?id=docencia
  - Páginas del manual: `man icpc`, o `konqueror man:icpc`.

# 5. PRIMERA TOMA DE CONTACTO CON ICC

## 5.5 - DEPURACIÓN DEL CÓDIGO

- Se emplea el programa depurador "idb" (`/opt/intel/Compiler/<versión>/bin/ia32/idb`), adjuntándole como argumento el fichero ejecutable a depurar. Condición previa para la depuración: Utilizar el argumento "-g", tanto en "icc" como en "icpc". Ejemplo:

```
[einiesta@localhost ejicc]$ ls -l
total 8
-rw-rw-r-- 1 einiesta einiesta 406 mar  9 12:47 suma.c
[einiesta@localhost ejicc]$ icc -g suma.c -o suma
[einiesta@localhost ejicc]$ ls -l
total 20
-rwxrwxr-x 1 einiesta einiesta 6720 mar  9 13:03 suma
-rw-rw-r-- 1 einiesta einiesta  406 mar  9 12:47 suma.c
[einiesta@localhost ejicc]$ idb ./suma
Intel(R) Debugger for applications running on IA-32, Version 10.0-29 , Build 20070405
------------------
object file name: ./suma
Reading symbols from /home/einiesta/am_0708/prac_0708/ejicc/suma...done.
(idb)
```

# 5. PRIMERA TOMA DE CONTACTO CON ICC

## DEPURACIÓN DEL CÓDIGO (continuación)

- Comandos interactivos de "idb":
  - r [<argumentos>]: Inicia la ejecución del programa.
  - b [<fich>]:<nlinea>: Detiene la ejecución en la línea <nlinea> (punto de ruptura).
  - b <función>: Detiene la ejecución en la primera línea de <función> (punto de ruptura).
  - info b: Muestra todos los puntos de ruptura.
  - delete b: Elimina todos los puntos de ruptura.
  - delete b <npr>: Elimina el punto de ruptura número <npr>.
  - c: Continúa una ejecución previamente detenida hasta el siguiente punto de ruptura, si lo hay, o hasta el final del programa, si no lo hay.
  - p <variable>: Muestra el valor de la variable <variable>.
  - p $<registro>: Muestra el valor del registro <registro> (ejemplo: `print $xmm0`).
  - n: Ejecuta la línea siguiente, y si esa línea corresponde a una función, ejecuta la función completa.
  - s: Ejecuta la línea siguiente, y si esa línea corresponde a una función, entra dentro de la función y se detiene en la primera de sus líneas.
  - backtrace: Muestra la pila de llamadas a funciones.

# 5. PRIMERA TOMA DE CONTACTO CON ICC

## 5.6 - EJEMPLO DE DEPURACIÓN

```
 1   #include <stdio.h>
 2
 3   int calcular_suma(int sumando_1,int sumando_2);
 4
 5   int main()
 6   {
 7     int a,b,s;
 8     printf("Introduzca sumando 1: ");scanf("%d",&a);
 9     printf("Introduzca sumando 2: ");scanf("%d",&b);
10     s=calcular_suma(a,b);
11     printf("Resultado de la suma: %d\n",s);
12     return(0);
13   }
14
15   int calcular_suma(int sumando_1,int sumando_2)
16   {
17     int resultado;
18     resultado=sumando_1+sumando_2;
19     return resultado;
20   }
```

# 5. PRIMERA TOMA DE CONTACTO CON ICC

## EJEMPLO DE DEPURACIÓN (continuación)

```
[einiesta@localhost ejicc]$ idb ./suma
Intel(R) Debugger for applications running on IA-32, Version 10.0-29 , Build 20070405
------------------
object file name: ./suma
Reading symbols from /home/einiesta/am_0708/prac_0708/ejicc/suma...done.
(idb) b 9
Breakpoint 1 at 0x804843c: file suma.c, line 9.
(idb) b calcular_suma
Breakpoint 2 at 0x80484a6: file suma.c, line 18.
(idb) r
Starting program: /home/einiesta/am_0708/prac_0708/ejicc/suma
Introduzca sumando 1: 5
Breakpoint 1, main () at suma.c:9
9         printf("Introduzca sumando 2: ");scanf("%d",&b);
(idb) p a
$1 = 5
(idb) p b
$2 = 13672436
(idb) n
```

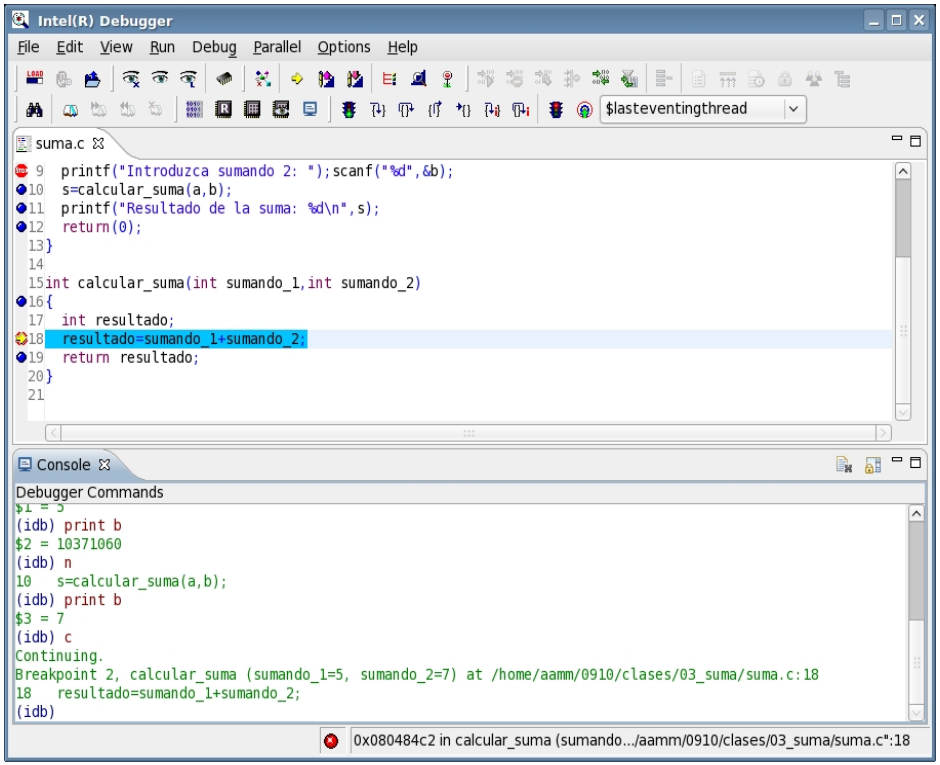
# 5. PRIMERA TOMA DE CONTACTO CON ICC

## EJEMPLO DE DEPURACIÓN (continuación)

```
Introduzca sumando 2: 7
10          s=calcular_suma(a,b);
(idb) print b
$3 = 7
(idb) c
Continuing.
Breakpoint 2, calcular_suma (sumando_1=5, sumando_2=7) at suma.c:18
18          resultado=sumando_1+sumando_2;
(idb) print resultado
$4 = 2198688
(idb) n
19          return resultado;
(idb) print resultado
$5 = 12
(idb) backtrace
#0  0x080484af in calcular_suma (sumando_1=5, sumando_2=7) at suma.c:19
#1  0x08048478 in main () at suma.c:10
(idb) c
Continuing.
Resultado de la suma: 12
Program exited normally.
```

# 5. PRIMERA TOMA DE CONTACTO CON ICC

## EJEMPLO DE DEPURACIÓN (interfaz gráfica)



# 5. PRIMERA TOMA DE CONTACTO CON ICC

## 5.7 - DEPURACIÓN "POSTMORTEM"

- Cuando en un programa se produzca una excepción de origen "desconocido", podemos, con "idb", resituar la ejecución justo en el instante anterior a la excepción, lo que nos permitirá, entre otras cosas, consultar el valor de las variables y determinar el contexto exacto bajo el que se ha producido el fallo. Pasos:

  - Forzar en las sesiones actuales la posibilidad de generar ficheros core de tamaño ilimitado: `ulimit -c unlimited` (recomendación: incluir esta orden en `$HOME/.bashrc`).
  - Compilar los fuentes con la opción "-g" (opción de depuración convencional).
  - Después del error, ejecutar "idb" pasándole como argumento el fichero core recién generado: `idb <programa> --core=<fichero_core>`

- La compilación definitiva del programa, una vez depurado, no debe llevarse a cabo con "-g" (ejecución mucho menos eficiente).

# 5. PRIMERA TOMA DE CONTACTO CON ICC

## 5.8 - ANÁLISIS DE LA EFICIENCIA

- Con "gprof" (`/usr/bin/gprof`) es posible conocer el desglose de tiempos consumidos por cada una de las funciones (o métodos) de un programa. Para ello, hay que seguir los pasos siguientes:

  - Compilar con la opción "-pg": `icc -pg -o <nombre_ejecutable> <nombre_fuente>.c`
  - Ejecutar el programa (se generará un fichero llamado "`gmon.out`").
  - Ejecutar `gprof ./<nombre_ejecutable>`
  - De la salida que produce la orden anterior, prestar especial atención a los campos "name" y "% time" (nombre de función y porcentaje de tiempo consumido).

- La compilación definitiva del programa, una vez depurado, no debe llevarse a cabo con "-pg" (aunque nos sirva para obtener tiempos relativos de funciones, resulta algo menos eficiente).

# 6. ENSAMBLADOR EN LÍNEA DENTRO DE CÓDIGO C/C++

## 6.1 - INTEGRACIÓN DE CÓDIGO ENSAMBLADOR GAS EN PROGRAMAS DE C/C++

- El código en ensamblador debe ir dispuesto del siguiente modo:

```
int main() {
  ...
  __asm__ volatile (
    <CÓDIGO GAS>
    <SECCIÓN E/S>
  );
  ...
}
```

- <CÓDIGO GAS>: Cada línea debe corresponderse con una instrucción del ensamblador GAS. Además, la línea debe ir entrecomillada y ha de terminar en "\n\t". Ejemplo:

```
"addl %%ebx, %%eax\n\t"
```

# 6. ENSAMBLADOR EN LÍNEA DENTRO DE CÓDIGO C/C++

## INTEGRACIÓN DE CÓDIGO ENSAMBLADOR GAS EN PROGRAMAS DE C/C++ (continuación)

- <SECCION E/S>: Se compone de tres partes,

  · :<salidas> Variables del fuente C/C++ cuyo valor va a ser alterado por el código ensamblador. Cada variable irá encerrada por "()", y precedida de "=<origen>", donde <origen> será (entre otras posibilidades) una "m" si la variable procede de memoria, y una "g" si procede de memoria, de un registro de la CPU o es un literal. Si hay más de una variable de salida, hay que emplear una "," para separar cada bloque de variable. Ejemplo ("resultado" es una variable declarada en C, fuera del bloque GAS):

```
: "=m" (resultado)
```

  · :<entradas> Variables del fuente C/C++ cuyo valor va a ser leido dentro del código ensamblador. Cada variable irá encerrada por "()", y precedida de "<origen>", donde <origen> será (entre otras posibilidades) una "m" si la variable procede de memoria, y una "g" si procede de memoria, de un registro de la CPU o es un literal. Si hay más de una variable de entrada, hay que emplear una "," para separar cada bloque de variable. Ejemplo ("sumando_1" y "sumando_2" son variables declarada en C):

```
: "m" (sumando_1), "m" (sumando_2)
```

# 6. ENSAMBLADOR EN LÍNEA DENTRO DE CÓDIGO C/C++

## INTEGRACIÓN DE CÓDIGO ENSAMBLADOR GAS EN PROGRAMAS DE C/C++ (continuación)

- <rec_pres> Conjunto de recursos preservados (memoria, registros...) que, después del bloque GAS, deben conservar el mismo valor que tenían antes de dicho bloque. Notación: “memory” (memoria), “<registro>” (registro de la CPU), “cc” (registro de estado). IMPORTANTE: Las variables de salida son “inmunes”, es decir, NUNCA se preservan (de lo contrario, nunca obtendríamos una salida).

  Ejemplo (forzar a que la memoria y los registros “eax” y “ebx” recuperen su valor anterior después de un bloque GAS):

```
: "memory", "%eax", "%ebx"
```

# 6. ENSAMBLADOR EN LÍNEA DENTRO DE CÓDIGO C/C++

## INTEGRACIÓN DE CÓDIGO ENSAMBLADOR GAS EN PROGRAMAS DE C/C++ (continuación)

- Otras consideraciones:
  - Los registros siempre van precedidos de "%%". Ejemplo: %%eax, %%edi, %%xmm0, etc.
  - Si una variable de salida, en lugar de ir con "=", va con "+", se convierte en variable de entrada/salida.
  - Las variables de entrada y las variables de salida son referenciadas dentro del bloque GAS mediante %0, %1, %2..., donde %0 es la primera variable de salida, %1 es la segunda variable de salida, y así hasta llegar a la última variable de salida, %n. A partir de ahí, %n+1 será la primera variable de entrada, %n+2 la segunda, etc.
- Más información:Archivo "asm-gcc-linea.pdf", en http://webs.um.es/einiesta/miwiki/doku.php?id=docencia

# 6. ENSAMBLADOR EN LÍNEA DENTRO DE CÓDIGO C/C++

## 6.2 - EJEMPLO: PROGRAMA "suma_inline" (código fuente)

```
#include <stdio.h>
int calcular_suma(int sumando_1,int sumando_2);

int main()
{
  int a,b,s;
  printf("Introduzca sumando 1: ");scanf("%d",&a);
  printf("Introduzca sumando 2: ");scanf("%d",&b);
  s=calcular_suma(a,b);
  printf("Resultado de la suma: %d\n",s);
        return(0);
}
int calcular_suma(int sumando_1,int sumando_2)
{
  int resultado;
  __asm__ volatile (
    "movl %1, %%eax\n\t"
    "movl %2, %%ebx\n\t"
    "addl %%ebx, %%eax\n\t"
    "movl %%eax,%0\n\t"
    : "=m" (resultado)
    : "m" (sumando_1), "m" (sumando_2)
    : "memory", "%eax", "%ebx"
  );
  return resultado;
}
```

# 6. ENSAMBLADOR EN LÍNEA DENTRO DE CÓDIGO C/C++

## EJEMPLO: PROGRAMA "suma_inline" (compilación, ejecución y depuración)

```
[einiesta@localhost prog_suma]$ icc -g suma_inline.c -o suma_inline
[einiesta@localhost prog_suma]$ ./suma_inline
Introduzca sumando 1: 7
Introduzca sumando 2: 4
Resultado de la suma: 11
[einiesta@localhost prog_suma]$ idb ./suma_inline
Intel(R) Debugger for applications running on IA-32, Version 10.0-29 , Build 20070405
object file name: ./suma_inline
Reading symbols from /home/einiesta/am_0708/prac_0708/ejinline/prog_suma/suma_inline...done.
(idb) b calcular_suma
Breakpoint 1 at 0x80484a6: file suma_inline.c, line 19.
(idb) r
Starting program: /home/einiesta/am_0708/prac_0708/ejinline/prog_suma/suma_inline
Introduzca sumando 1: 7
Introduzca sumando 2: 4
Breakpoint 1, calcular_suma (sumando_1=7, sumando_2=4) at suma_inline.c:19
19        __asm__ volatile (
(idb) print $ebx
$1 = 13672436
(idb) n
30        return resultado;
(idb) print $ebx
$2 = 4
(idb) c
Continuing.
Resultado de la suma: 11
```

# 7. EJEMPLO BÁSICO DE USO DE EXTENSIONES MULTIMEDIA

## 7.1 - CÓDIGO FUENTE DE "sumasse" (SUMA DE DOS VECTORES CON SSE)

```
/* Ejemplo de suma de dos vectores de 4 floats cada uno (1 float = 32 bits,
IEEE 754 de simple precisión), utilizando extensiones SSE. */

#include <stdio.h>
#include <xmmintrin.h>

// Suma de dos vectores utilizando SSE.
// - Parámetros de entrada: 2 vectores de 4 floats, en formato empaquetado (128 bits).
// - Parámetros de salida: El vector de los 4 floats resultado, en formato empaquetado (128 bits).
__m128 sumar_vectores(__m128 v01, __m128 v02);

// Función principal.

int main(void)
{
// Declaramos punteros a las áreas de memoria en donde se alojarán los vectores
// operandos (v01 y v02) y el vector resultado (v03).
float *v01, *v02, *v03;
// Buffers para guardar los operandos y el resultado en formato empaquetado.
__m128 buf_v01, buf_v02, buf_v03;
```

# 7. EJEMPLO BÁSICO DE USO DE EXTENSIONES MULTIMEDIA

## CÓDIGO FUENTE DE "sumasee" (continuación)

```
// Reserva de la memoria necesaria para las áreas v01, v02 y v03.
// Importante: El valor "16" con el que es instanciado el segundo parámetro
// de "_mm_malloc" nos asegura que cada área de memoria reservada empieza en
// una dirección que sea múltiplo de 16 bytes. Ello optimiza el acceso
// a la memoria, ya que los accesos a memoria realizados por instrucciones
// SSE son accesos alineados a 16 bytes.
v01 = (float *) _mm_malloc(4*sizeof(float),16);
v02 = (float *) _mm_malloc(4*sizeof(float),16);
v03 = (float *) _mm_malloc(4*sizeof(float),16);

// Inicialización de sumandos.
v01[0]=1;v01[1]=2;v01[2]=3;v01[3]=4;
v02[0]=5;v02[1]=6;v02[2]=7;v02[3]=8;

// Introducción de los sumandos en los buffers (la instrucción SSE que
// utilizaremos a continuación, "addps", exige que los operandos estén
// en estructuras "__m128").
buf_v01=_mm_set_ps(v01[3],v01[2],v01[1],v01[0]);
buf_v02=_mm_set_ps(v02[3],v02[2],v02[1],v02[0]);
```

# 7. EJEMPLO BÁSICO DE USO DE EXTENSIONES MULTIMEDIA

## CÓDIGO FUENTE DE "sumasee" (continuación)

```
// Suma de los vectores.
buf_v03=sumar_vectores(buf_v01,buf_v02);

// Desempaquetamiento del resultado.
_mm_store_ps(v03,buf_v03);

// Exhibición de resultados.
printf("%f %f %f %f\n",v03[0],v03[1],v03[2],v03[3]);

// Liberación de la memoria previamente reservada.
_mm_free(v01);
_mm_free(v02);
_mm_free(v03);
return 0;
}
```

# 7. EJEMPLO BÁSICO DE USO DE EXTENSIONES MULTIMEDIA

## CÓDIGO FUENTE DE "sumasee" (continuación)

```
// Suma de dos vectores utilizando SSE.
// - Parámetros de entrada: 2 vectores de 4 floats,
// en formato empaquetado (128 bits).
// - Parámetros de salida: El vector de los 4 floats
// resultado, en formato empaquetado (128 bits).

__m128 sumar_vectores(__m128 v01, __m128 v02)
{
__m128 resultado=_mm_set_ps(0,0,0,0); // Inicializamos a 0 el resultado.
// Código inline: Suma vectorial mediante la instrucción SSE "addps"
__asm__ volatile (
"movaps %1,%%xmm0\n\t" // xmm0 = v01
"movaps %2,%%xmm1\n\t" // xmm1 = v02
"addps %%xmm1,%%xmm0\n\t" // xmm0 = xmm0 + xmm1
"movaps %%xmm0,%0\n\t" // resultado = xmm0
: "=m" (resultado)
: "m" (v01), "m" (v02)
: "memory"
);
return resultado;
}
```

# 7. EJEMPLO BÁSICO DE USO DE EXTENSIONES MULTIMEDIA

## 7.2 - EJECUCIÓN DE "sumasee"

```
[aamm@localhost 07_sumasse]$ ls -l
total 4
-rw-rw-r-- 1 aamm aamm 2907 mar 14 14:58 sumasse.c
[aamm@localhost 07_sumasse]$ icc -o sumasse sumasse.c
[aamm@localhost 07_sumasse]$ ls -l
total 28
-rwxrwxr-x 1 aamm aamm 20885 mar 14 14:58 sumasse
-rw-rw-r-- 1 aamm aamm  2907 mar 14 14:58 sumasse.c
[aamm@localhost 07_sumasse]$ ./sumasse
6.000000 8.000000 10.000000 12.000000
```

# 8. USO DE LAS EXTENSIONES MULTIMEDIA DESDE GCC

## 8.1 – COMPILACIÓN DE FUENTES CON GCC

- El conjunto de compiladores GCC ("Global Compiler Collection") constituyen un estándar de desarrollo dentro de la programación bajo Linux (de hecho, suelen venir incorporados en todas las distribuciones).

- De entre todos los compiladores de GCC ("gcc" para C, "g++" para C++, "gcj" para Java, etc) dos de ellos, el "gcc" y el "g++", pueden representar una alternativa al "Intel icc" y al "Intel icpc", respectivamente (SI BIEN EN ESTAS PRÁCTICAS SE USARÁ, EN TODO MOMENTO, INTEL C++).

- Los ficheros de cabecera necesarios para compilar con "gcc" o "g++" fuentes que hagan uso de extensiones multimedia (`mmintrin.h`, `xmmintrin.h`, `emmintrin.h`, etc) se denominan igual que sus homólogos de Intel C++. En el caso de GCC, dichos ficheros de cabecera están (para un Fedora 8) en `/usr/lib/gcc/i386-redhat-linux/4.1.2/include`.

- Los mismos fuentes que hayan sido compilados con Intel C++ pueden ser recompilados, sin alterarlos, con GCC (comprobado con la versión 4.1.2 de gcc).

# 8. USO DE LAS EXTENSIONES MULTIMEDIA DESDE GCC

## COMPILACIÓN DE FUENTES CON GCC (continuación)

- La compilación con "gcc" (C) o "g++" (C++), cuando el código fuente incluya extensiones multimedia, requiere la participación de un parámetro adicional que establezca cuáles son exactamente las extensiones que se quieren utilizar:

  - -mmmx: Extensiones MMX.
  - -msse: Extensiones SSE.
  - -msse2: Extensiones SSE2.
  - -msse3: Extensiones SSE3.
  - Etc (ver hoja del manual de "gcc" o "g++").

# 8. USO DE LAS EXTENSIONES MULTIMEDIA DESDE GCC

## 8.2 – EJEMPLO DE COMPILACIÓN CON GCC (código fuente sumasse.c, ver apartado anterior)

```
[aamm@localhost 08_sumasse_gcc]$ ls ../07_sumasse/*.c
../07_sumasse/sumasse.c
[aamm@localhost 08_sumasse_gcc]$ gcc -msse -o sumasse_gcc ../07_sumasse/sumasse.c
[aamm@localhost 08_sumasse_gcc]$ ls
sumasse_gcc
[aamm@localhost 08_sumasse_gcc]$ ./sumasse_gcc
6.000000 8.000000 10.000000 12.000000
[aamm@localhost 08_sumasse_gcc]$ gcc --version
gcc (GCC) 4.1.2 20070925 (Red Hat 4.1.2-33)
Copyright (C) 2006 Free Software Foundation, Inc.
Esto es software libre; vea el código para las condiciones de copia.  NO hay
garantía; ni siquiera para MERCANTIBILIDAD o IDONEIDAD PARA UN PROPÓSITO EN
PARTICULAR
```

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

## 9.1 - MODOS DE PROGRAMACIÓN CON EXTENSIONES MULTIMEDIA

- Método 1: Utilizando directamente el ensamblador en línea de GAS (integrado en los fuentes C/C++).

- Método 2: Recurriendo a las funciones "intrínsecas" incluidas en las librerías del Intel C++. Los ficheros de cabecera en donde se definen estas intrínsecas están en "`/opt/intel/Compiler/<version>/include`", y son: "`mmintrin.h`", "`xmmintrin.h`", "`emmintrin.h`" y "`pmmintrin.h`". Con este método se pueden utilizar, indistintamente, el compilador "icc" o el "icpc" (fuentes ".c" y ".cpp", respectivamente).

- Método 3: Apoyándose en las clases C++ incluidas en las librerías del Intel C++. Los ficheros de cabecera asociados se encuentran en "`/opt/intel/Compiler/<version>/include`", y son: "`ivec.h`", "`fvec.h`" y "`dvec.h`". Con este método sólo se puede usar el compilador "icpc" (se programa con clases C++).

  Desde el punto de vista de la eficiencia, los tres métodos son equivalentes. El primer método sirve para conocer mejor las instrucciones MMX, SSE, SSE2 y SSE3. El segundo y tercer métodos, una vez conocidas esas instrucciones, facilitan la tarea de programación, al no ser ya necesario recurrir a código en ensamblador.

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

## 9.2 - INSTRUCCIONES MMX

- Contexto: A partir del Pentium MMX (año 1997). Registros MM0...MM7, de 64 bits. Sólo permiten operaciones con enteros, en grupos empaquetados de 8, 16 o 32 bits. Las extensiones MMX, aparte de sus propios registros, también utilizan registros de la Unidad de Punto Flotante Escalar (FPU); por ello, cuando las hayamos utilizado es conveniente reiniciar dichos registros FPU con la instrucción "emms".

- 57 instrucciones: "paddb", "paddw", "paddd", "psubb", etc (casi todas empiezan por "p").

- Programación con intrínsecas: Las declaraciones están en "mmintrin.h"; las implementaciones están en los archivos ".a" y ".so" de "`/opt/intel/Compiler/<versión>/lib/ia32`" (ver LD_LIBRARY_PATH).

- Programación con clases C++: Las declaraciones de las clases están en "ivec.h"; las implementaciones de los métodos de esas clases se apoyan en llamadas a las intrínsecas de "mmintrin.h". Algunas clases disponibles:

  - Is8vec8, Iu8vec8: Para operar con 8 grupos de enteros de 8 bits, con signo y sin signo, respectivamente.
  - Is16vec4, Iu16vec4: Para operar con 4 grupos de enteros de 16 bits, con signo y sin signo, respectivamente.
  - Is32vec2, Iu32vec2: Para operar con 2 grupos de enteros de 32 bits, con signo y sin signo, respectivamente.

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

```
#include <stdio.h>
#include <mmintrin.h>

/* Ejemplo de suma de dos vectores de 4 enteros sin signo. Cada entero es un valor almacenable en 2 bytes.
   La suma se realizará con extensiones MMX invocadas desde intrínsecas. */

int main(void)
{
  // Declaración de los vectores sumandos, "v01" y "v02", y del vector resultado, "vsuma".
  short int v01[4]={1,2,3,4}, v02[4]={5,6,7,8};
  short int vsuma[4];
  // Empaquetamiento de los operandos.
  __m64 buf_v01=_mm_set_pi16(v01[3],v01[2],v01[1],v01[0]); // Sufijo "pi16" = "packed
  __m64 buf_v02=_mm_set_pi16(v02[3],v02[2],v02[1],v02[0]); // integer 16 bits".
  __m64 buf_vsuma;
  // Suma vectorial.
  buf_vsuma=_m_paddw(buf_v01,buf_v02); // Ejecución "encubierta" de "paddw".
  // Extracción de resultados.
  vsuma[0]=_m_to_int(_m_pand(buf_vsuma,_mm_set_pi16(0,0,0,65535)));               // "_m_to_int" espera
  vsuma[1]=_m_to_int(_m_pand(_m_psrlqi(buf_vsuma,16),_mm_set_pi16(0,0,0,65535))); // encontrar un único
  vsuma[2]=_m_to_int(_m_pand(_m_psrlqi(buf_vsuma,32),_mm_set_pi16(0,0,0,65535))); // operando empaquetado.
  vsuma[3]=_m_to_int(_m_pand(_m_psrlqi(buf_vsuma,48),_mm_set_pi16(0,0,0,65535)));
  // Instrucción EMMS: "reset" de los registros de la FPU (prevenir futuros errores en operaciones escalares).
  _m_empty();
  printf("Resultado: %d %d %d %d\n",vsuma[0],vsuma[1],vsuma[2],vsuma[3]);
  return 0; }
```

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

```
#include <stdio.h>
#include <ivec.h>

/* Ejemplo de suma de dos vectores de 4 enteros sin signo. Cada entero es un valor almacenable en 2 bytes.
   La suma se realizará con extensiones MMX invocadas desde clases C++. */

int main(void)
{
  // Declaración de los vectores sumandos, "v01" y "v02", y del vector resultado, "vsuma".
  Iu16vec4 v01=Iu16vec4(4,3,2,1);
  Iu16vec4 v02=Iu16vec4(8,7,6,5);
  Iu16vec4 vsuma;
  // Variable para la extracción del resultado.
  short int vsuma_no_packed[4];

  // Suma vectorial.
  vsuma=v01+v02; // Ejecución "encubierta" de "paddw".

  // Extracción de resultados.
  vsuma_no_packed[0]=_m_to_int(_m_pand(vsuma,_mm_set_pi16(0,0,0,65535)));
  vsuma_no_packed[1]=_m_to_int(_m_pand(_m_psrlqi(vsuma,16),_mm_set_pi16(0,0,0,65535)));
  vsuma_no_packed[2]=_m_to_int(_m_pand(_m_psrlqi(vsuma,32),_mm_set_pi16(0,0,0,65535)));
  vsuma_no_packed[3]=_m_to_int(_m_pand(_m_psrlqi(vsuma,48),_mm_set_pi16(0,0,0,65535)));
  printf("Resultado: %d %d %d %d\n",vsuma[0],vsuma[1],vsuma[2],vsuma[3]);
  return 0;
}
```

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

## 9.3 - INSTRUCCIONES SSE

- Contexto: A partir del Pentium 3 (año 1999). Registros XMM0...XMM7, de 128 bits. Sólo permiten operaciones con valores en punto flotante de simple precisión (IEEE754), en 4 grupos empaquetados de 32 bits. El almacenamiento en memoria de estos valores debe hacerse respetando un alineamiento de 16 bytes.

- 70 instrucciones: “addps”, “subps”, “mulps”, “divps”, “movaps”, etc.

- Programación con intrínsecas: Las declaraciones están en “xmmintrin.h”; las implementaciones están en los archivos “.a” y “.so” de “`/opt/intel/Compiler/<versión>/lib/ia32`” (ver LD_LIBRARY_PATH).

- Programación con clases C++: Las declaraciones de las clases están en “fvec.h”; las implementaciones de los métodos de esas clases se apoyan en llamadas a las intrínsecas de “xmmintrin.h”. La clase de “fvec.h” más importante es “F32vec4”, diseñada para realizar operaciones aritméticas y lógicas simultáneas entre 4 grupos de valores en punto flotante representados en simple precisión (32 bits).

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

## INSTRUCCIONES SSE (continuación)

- Alineamiento de los arrays en punto flotante: Las instrucciones SSE y posteriores (SSE2, SSE3, etc) trabajan con operandos alineados a 16 bytes. Cuando utilizamos instancias de clases C++ (F32vec4, F64vec2...) ese alineamiento ya nos los asegura el propio compilador.

- Sin embargo, ése no es el caso de los arrays en punto flotante. Por ello, cuando queramos emplear uno de estos arrays para obtener el contenido desempaquetado de un vector (__m128, __m128d, __m128i, etc), hemos de forzar dicho alineamiento (de lo contrario, puede producirse una excepción en tiempo de ejecución).

- Si el array es creado en tiempo de ejecución, establecemos el alineamiento mediante la instrucción "`_mm_malloc`" (para la liberación posterior, utilizar "`_mm_free`").

- Si el espacio necesario para el array es reservado en tiempo de compilación, es preciso anteponer el prefijo "`__declspec(align(16))`" antes de la declaración del array.

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

## INSTRUCCIONES SSE (continuación)

- Ejemplo de reserva dinámica de memoria:

```
float *arrayNoPacked = (float *) _mm_malloc(4*sizeof(float),16);
...
_mm_store_ps(arrayNoPacked, vector);
...
_mm_free(arrayNoPacked);
```

- Ejemplo de reserva estática de memoria:

```
__declspec (align(16)) float arrayNoPacked[4];
...
_mm_store_ps(arrayNoPacked, vector);
...
```

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

```
#include <stdio.h>
#include <xmmintrin.h>

/* Ejemplo de suma de dos vectores de 4 float. Cada float es un valor almacenable en 4 bytes (IEEE 754
   simple precisión). La suma se realizará con extensiones SSE invocadas desde intrínsecas. */

int main(void)
{
  // Declaración de los vectores sumandos, "v01" y "v02", y del vector resultado, "vsuma".
  float v01[4]={1.1,2.2,3.3,4.4};
  float v02[4]={5.5,6.6,7.7,8.8};
  __declspec (align(16)) float vsuma[4];

  // Empaquetamiento de los operandos.
  __m128 buf_v01=_mm_set_ps(v01[3],v01[2],v01[1],v01[0]); // Sufijo "ps" = "packed single".
  __m128 buf_v02=_mm_set_ps(v02[3],v02[2],v02[1],v02[0]);
  __m128 buf_vsuma;

  // Suma vectorial.
  buf_vsuma=_mm_add_ps(buf_v01,buf_v02); // Ejecución "encubierta" de "addps".

  // Extracción de resultados.
  _mm_store_ps(vsuma,buf_vsuma);
  printf("Resultado: %f %f %f %f\n",vsuma[0],vsuma[1],vsuma[2],vsuma[3]);
  return 0;
}
```

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

```
#include <stdio.h>
#include <fvec.h>

/* Ejemplo de suma de dos vectores de 4 float. Cada float es un valor almacenable en 4 bytes (IEEE 754
   simple precisión). La suma se realizará con extensiones SSE invocadas desde clases C++. */

int main(void)
{
  // Declaración de los vectores sumandos, "v01" y "v02", y del vector resultado, "vsuma".
  F32vec4 v01, v02, vsuma;
  v01=F32vec4(4.4,3.3,2.2,1.1);
  v02=F32vec4(8.8,7.7,6.6,5.5);

  // Variable para la extracción del resultado.
  __declspec (align(16)) float vsuma_no_packed[4];

  // Suma vectorial.
  vsuma=v01+v02;

  // Extracción de resultados.
  _mm_store_ps(vsuma_no_packed,vsuma);
  printf("Resultado: %f %f %f %f\n",vsuma_no_packed[0],vsuma_no_packed[1],
         vsuma_no_packed[2],vsuma_no_packed[3]);
  return 0;
}
```

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

## 9.4 - INSTRUCCIONES SSE2

- Contexto: A partir del Pentium 4 (año 2000). Registros XMM0...XMM7, de 128 bits (idénticos a los de SSE). Permiten operaciones con valores en punto flotante de doble precisión (IEEE 754), siendo los operandos grupos de 2 "doubles" (64 bits cada uno). SSE2 también aporta instrucciones para números enteros; posibilidades: operandos formados por 16 grupos de 8 bits, o formados por 8 grupos de 16 bits, o por 4 grupos de 32 bits. El almacenamiento en memoria de estos valores debe hacerse respetando un alineamiento de 16 bytes (igual que en SSE).

- 144 instrucciones: a) Operaciones entre valores en punto flotante: "addpd", "subpd", "mulpd", "divpd", etc; b) Operaciones entre valores enteros: "addepi8", "addepi16", "addepi32", "subepi8", etc; c) Otras: "movapd"...

- Programación con intrínsecas: Las declaraciones están en "emmintrin.h"; las implementaciones están en los archivos ".a" y ".so" de "`/opt/intel/Compiler/<versión>/lib/ia32`" (ver variable de entorno LD_LIBRARY_PATH).

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

## INSTRUCCIONES SSE2 (continuación)

- Programación con clases C++: Las declaraciones de las clases están en "dvec.h"; las implementaciones de los métodos de esas clases se apoyan en llamadas a las intrínsecas de "emmintrin.h". Algunas de las clases más importantes son:

  - Para operaciones entre valores en punto flotante: F64vec2.

  - Para operaciones entre valores enteros: Is8vec16, Iu8vec16, Is16vec8, Iu16vec8, Is32vec4, Iu32vec4.

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

```
#include <stdio.h>
#include <emmintrin.h>

/* Ejemplo de suma de dos vectores de 2 componentes double. Cada double es un valor almacenable en 8 bytes
   (IEEE 754 doble precisión). La suma se realizará con extensiones SSE2 invocadas desde intrínsecas. */

int main(void)
{
  // Declaración de los vectores sumandos, "v01" y "v02", y del vector resultado, "vsuma".
  double v01[2]={1.1,2.2};
  double v02[2]={3.3,4.4};
  __declspec (align(16)) double vsuma[2];

  // Empaquetamiento de los operandos.
  __m128d buf_v01=_mm_set_pd(v01[1],v01[0]); // Sufijo "pd" = "packed double".
  __m128d buf_v02=_mm_set_pd(v02[1],v02[0]);
  __m128d buf_vsuma;

  // Suma vectorial.
  buf_vsuma=_mm_add_pd(buf_v01,buf_v02); // Ejecución "encubierta" de "addpd".

  // Extracción de resultados.
  _mm_store_pd(vsuma,buf_vsuma);
  printf("Resultado: %f %f\n",vsuma[0],vsuma[1]);
  return 0;
}
```

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

```
#include <stdio.h>
#include <dvec.h>

/* Ejemplo de suma de dos vectores de 2 componentes double. Cada double es un valor almacenable en 8 bytes
   (IEEE 754 doble precisión). La suma se realizará con extensiones SSE2 invocadas desde clases C++. */

int main(void)
{
  // Declaración de los vectores sumandos, "v01" y "v02", y del vector resultado, "vsuma".
  F64vec2 v01, v02, vsuma;
  v01=F64vec2(2.2,1.1);
  v02=F64vec2(6.6,5.5);
  // Variable para la extracción del resultado.
  __declspec (align(16)) double vsuma_no_packed[2];

  // Suma vectorial.
  vsuma=v01+v02;

  // Extracción de resultados.
  _mm_store_pd(vsuma_no_packed,vsuma);
  printf("Resultado: %f %f\n",vsuma_no_packed[0],vsuma_no_packed[1]);
  return 0;
}
```

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

```
#include <stdio.h>
#include <emmintrin.h>

/* Ejemplo de suma de dos vectores de 4 componentes enteros. Cada entero es un valor almacenable
   en 4 bytes (representación en complemento a 2; rango de representación: [-2147483648,2147483647]).
   La suma se realizará con extensiones SSE2 invocadas desde intrínsecas. */

int main(void)
{
  // Declaración de los vectores sumandos, "v01" y "v02", y del vector resultado, "vsuma".
  int v01[4]={1000000000,1000000000,1000000000,-2000000000};
  int v02[4]={1,2,3,4};
  __declspec (align(16)) int vsuma[4];

  // Empaquetamiento de los operandos.
  __m128i buf_v01=_mm_set_epi32(v01[3],v01[2],v01[1],v01[0]);
  __m128i buf_v02=_mm_set_epi32(v02[3],v02[2],v02[1],v02[0]);
  __m128i buf_vsuma;

  // Suma vectorial.
  buf_vsuma=_mm_add_epi32(buf_v01,buf_v02); // Ejecución "encubierta" de "addepi32".
```

## 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

```
  // Extracción de resultados.
  // NOTA: De cada grupo de 32 bits hay que ir sacando la parte baja (los 16 bits menos
  // significativos) y la parte alta (los 16 bits más significativos), y multiplicar este
  // segundo valor por 2^16 (65536).
  vsuma[0]=_mm_extract_epi16(buf_vsuma,0)+65536*_mm_extract_epi16(buf_vsuma,1);
  vsuma[1]=_mm_extract_epi16(buf_vsuma,2)+65536*_mm_extract_epi16(buf_vsuma,3);
  vsuma[2]=_mm_extract_epi16(buf_vsuma,4)+65536*_mm_extract_epi16(buf_vsuma,5);
  vsuma[3]=_mm_extract_epi16(buf_vsuma,6)+65536*_mm_extract_epi16(buf_vsuma,7);

  printf("Resultado: %d %d %d %d\n",vsuma[0],vsuma[1],vsuma[2],vsuma[3]);
  return 0;
}
```

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

```
#include <stdio.h>
#include <dvec.h>

/* Ejemplo de suma de dos vectores de 4 componentes enteros. Cada entero es un valor almacenable en 4 bytes
   (representación en complemento a 2; rango de representación: [-2147483648,2147483647]).
   La suma se realizará con extensiones SSE2 invocadas desde clases C++. */

int main(void)
{
  // Declaración de los vectores sumandos, "v01" y "v02", y del vector resultado, "vsuma".
  Is32vec4 v01, v02, vsuma;
  __declspec (align(16)) int vsuma_no_packed[4];
  v01=Is32vec4(-2000000000,1000000000,1000000000,1000000000);
  v02=Is32vec4(4,3,2,1);

  // Suma vectorial.
  vsuma=v01+v02; // Ejecución "encubierta" de "addepi32".

  // Extracción de resultados.
  vsuma_no_packed[0]=_mm_extract_epi16(vsuma,0)+65536*_mm_extract_epi16(vsuma,1);
  vsuma_no_packed[1]=_mm_extract_epi16(vsuma,2)+65536*_mm_extract_epi16(vsuma,3);
  vsuma_no_packed[2]=_mm_extract_epi16(vsuma,4)+65536*_mm_extract_epi16(vsuma,5);
  vsuma_no_packed[3]=_mm_extract_epi16(vsuma,6)+65536*_mm_extract_epi16(vsuma,7);
  printf("Resultado: %d %d %d %d\n",vsuma_no_packed[0],vsuma_no_packed[1],
                                    vsuma_no_packed[2],vsuma_no_packed[3]);
  return 0; }
```

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

## 9.5 - INSTRUCCIONES SSE3

- Contexto: A partir del Intel Core Duo (año 2005). Registros XMM0...XMM7, de 128 bits (idénticos a los de SSE). Permiten operaciones con valores en punto flotante, tanto de simple como de doble precisión (grupos de 2 floats de 64 bits en el primer caso, y grupos de 4 floats de 32 bits en el segundo). El almacenamiento en memoria de estos valores debe hacerse respetando un alineamiento de 16 bytes (igual que en SSE).

- 13 instrucciones: Combinaciones de sumas y restas (“addsubpd”, “addsubps”), sumas y restas horizontales (“haddpd”, “hsubpd”, “haddps”, “hsubps”), etc.

- Programación con intrínsecas: Las declaraciones están en “pmmintrin.h”; las implementaciones están en los archivos “.a” y “.so” de “`/opt/intel/Compiler/<versión>/lib/ia32`” (ver variable de entorno LD_LIBRARY_PATH).

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

## 9.6 - INSTRUCCIONES SSSE3

- Contexto: A partir del Intel Core 2 Duo (año 2006). Registros XMM0...XMM7, de 128 bits (idénticos a los de SSE). Permiten operaciones con enteros de tipo byte, palabra y doble palabra (8, 16 y 32 bits, respectivamente). Almacenamiento en memoria respetando un alineamiento de 16 bytes (igual que en SSE).

- 16 instrucciones, de tipo básicamente aritmético: phaddw ("packed horizontal add words"), phaddd (packed horizontal add doublewords"), phsubw (similar a phaddw, pero para la resta), phsubd (similar a phaddd, pero para la resta), etc.

- Programación con intrínsecas: Las declaraciones están en "tmmintrin.h"; las implementaciones están en los archivos ".a" y ".so" de "`/opt/intel/Compiler/<versión>/lib/ia32`" (ver variable de entorno LD_LIBRARY_PATH).

# 9. RECURSOS DEL INTEL C++ PARA PROGRAMAR CON EXTENSIONES MULTIMEDIA

```
#include <stdio.h>
#include <tmmintrin.h>

/* Ejemplo de resta vectorial horizontal (componentes de tipo word -16 bits-). Extensiones SSSE3. */

int main(void)
{
  // Declaración de los operandos.
  short int v01[8]={1,2,4,8,16,32,64,128};
  short int v02[8]={10,20,20,10,0,0,-3,-1};
  __declspec (align(16)) short int vresta[8];
  // Resultado esperado: (1-2, 4-8, 16-32, 64-128, 10-20, 20-10, 0-0, -3-(-1)) =
  //                     (-1, -4, -16, -64, -10, 10, 0, -2)
  // Empaquetamiento de los operandos.
  __m128i buf_v01=_mm_set_epi16(v01[7],v01[6],v01[5],v01[4],v01[3],v01[2],v01[1],v01[0]);
  __m128i buf_v02=_mm_set_epi16(v02[7],v02[6],v02[5],v02[4],v02[3],v02[2],v02[1],v02[0]);
  __m128i buf_vresta;
  // Resta horizontal.
  buf_vresta=_mm_hsub_epi16(buf_v01,buf_v02); // Ejecución "encubierta" de "phsubw"
                                              // (Packed Horizontal SUBtract for Words)
  // Extracción de resultados.
  vresta[0]=_mm_extract_epi16(buf_vresta,0);vresta[1]=_mm_extract_epi16(buf_vresta,1);
  vresta[2]=_mm_extract_epi16(buf_vresta,2);vresta[3]=_mm_extract_epi16(buf_vresta,3);
  vresta[4]=_mm_extract_epi16(buf_vresta,4);vresta[5]=_mm_extract_epi16(buf_vresta,5);
  vresta[6]=_mm_extract_epi16(buf_vresta,6);vresta[7]=_mm_extract_epi16(buf_vresta,7);

  printf("Resultado: %d %d %d %d %d %d %d %d\n",
         vresta[0], vresta[1], vresta[2], vresta[3], vresta[4], vresta[5], vresta[6], vresta[7]);
  return 0; }
```

# 10. COMPILACIÓN MULTIFUNCIONAL

## 10.1 - UTILIDAD MAKE

- "`make`" es una herramienta de Linux. Sirve para generar un archivo ejecutable a partir de otro/s archivo/s fuente/s. Su uso más frecuente es la compilación de código fuente en C o C++.

- Se basa en un fichero de configuración denominado "`Makefile`". La estructura de este fichero es la siguiente (IMPORTANTE: Usar el tabulador, y no espacios en blanco, para introducir los sangrados de los comandos):

```
[<VARIABLES>]

objetivo_1:  [<archivo_entrada11> <archivo_entrada12> ... ]
             <comando11>
             <comando12>
             ...
objetivo_2:  [<archivo_entrada21> <archivo_entrada22> ... ]
             <comando21>
             <comando22>
             ...
```

## 10. COMPILACIÓN MULTIFUNCIONAL

- Consideraciones:

  - Cuando se ejecuta `make` sin argumentos, se procesa la sección de `Makefile` correspondiente al primer objetivo. Si se adjunta algún argumento, éste debe ser el nombre de uno de los objetivos de `Makefile`.
  - Al ejecutar `make`, sólo se ejecutarán los comandos del objetivo seleccionado si se detecta algún cambio en los archivos de entrada.
  - Los objetivos pueden tener relaciones de dependencia: Si algún comando de un objetivo altera un fichero de entrada de otro objetivo, automáticamente se ejecutarán los comandos de este segundo objetivo.
  - Las variables aluden normalmente a algún comando o programa. Se declaran mediante identificadores en mayúscula, y son leídas en `Makefile` entre paréntesis y con el prefijo "`$`". Ejemplo: `$(CC)`.
  - Makefile ya cuenta con algunas variables predeclaradas. Ejemplos: `CC` (compilador; su valor por defecto es "`cc`" -enlace simbólico al gcc-), `RM` (alias del comando "`rm -f`"), etc.
  - Los archivos de entrada no son obligatorios. Si no se indican, los comandos del objetivo siempre se ejecutan (no es necesario que cambie un fichero de entrada, puesto que no hay ficheros de entrada).
  - `make -f <fichero_Makefile>` hace que el fichero de configuración consultado por `make` sea `<fichero_Makefile>`, en vez de ser `Makefile`.

## 10. COMPILACIÓN MULTIFUNCIONAL

- Ejemplo:

```
[aamm@localhost 17_wt_sec]$ cat Makefile
CC = icc

# Compilación normal.
wt:             wt.c
                $(CC) -Wall -o wt wt.c
# Modo depuración.
debug:          wt.c
                icc -Wall -g -o wt wt.c
# Modo "profiling".
prof:           wt.c
                icc -Wall -pg -o wt wt.c
# Eliminación de todo lo que no sean fuentes.
clean:
        $(RM) wt
        $(RM) *~
        $(RM) gmoun.out
```

## 10. COMPILACIÓN MULTIFUNCIONAL

```
[aamm@localhost 17_wt_sec]$ make
icc -Wall -o wt wt.c
[aamm@localhost 17_wt_sec]$ make
make: `wt' está actualizado.
[aamm@localhost 17_wt_sec]$ ls -l wt
-rwxrwxr-x 1 aamm aamm 20324 mar 27 19:25 wt
[aamm@localhost 17_wt_sec]$ make debug
icc -Wall -g -o wt wt.c
[aamm@localhost 17_wt_sec]$ ls -l wt
-rwxrwxr-x 1 aamm aamm 5504 mar 27 19:25 wt
[aamm@localhost 17_wt_sec]$ make clean
rm -f wt
rm -f *~
rm -f gmoun.out
[aamm@localhost 17_wt_sec]$ make -f Makefile_gcc
cc -Wall -o wt wt.c
[aamm@localhost 17_wt_sec]$ ls -l wt
-rwxrwxr-x 1 aamm aamm 4788 mar 27 19:26 wt
```

# 10. COMPILACIÓN MULTIFUNCIONAL

## 10.2 - USO DE DIRECTIVAS EN EL CÓDIGO FUENTE

- La inclusión de directivas en el código fuente permite seleccionar qué líneas de un fichero fuente son tenidas en cuenta para la compilación.

- Ejemplo:

```
// Código que siempre se compila.
// ...
// Código que únicamente se compila si hemos activado la directiva   MOSTRAR  .
#ifdef MOSTRAR
printf("ENTRADA:");
for (i=0; i<longVector; i++) printf("%8.2f ", vector[i]);
#endif
// Código que siempre se compila.
...
```

- Así, podemos cambiar la funcionalidad del ejecutable sin por ello tener que alterar su código fuente.

## 10. COMPILACIÓN MULTIFUNCIONAL

- Tres formas de utilización de directivas:

```
#ifdef <DIRECTIVA>
// Código que se compila si está activa la directiva.
#endif


#ifndef <DIRECTIVA>
// Código que se compila si no está activa la directiva.
#endif


#ifdef <DIRECTIVA>
// Código que se compila si está activa la directiva.
#else
// Código que se compila si no está activa la directiva.
#endif
```

## 10. COMPILACIÓN MULTIFUNCIONAL

- Dos formas de definir directivas:
  - En el propio código fuente: `#define MOSTRAR`
  - Como argumento durante la compilación: `icc -D MOSTRAR -Wall -o wt wt.c`
- Para los ficheros ".h", importante prevenir (con una directiva) la doble inclusión del ".h" en un proyecto:
  - Ejemplo ("`constantes.h`"):

```
#ifndef __CONSTANTES
#define __CONSTANTES

#define TRUE 1
#define FALSE 1
// Resto del código...

#endif
```

# 10. COMPILACIÓN MULTIFUNCIONAL

## 10.3 - INTERFAZ EN LÍNEA DE COMANDOS

- Una vez definida la funcionalidad del programa, en ciertos casos puede ser interesante suministrarle a éste parámetros NO con E/S interactiva (formulando preguntas, por ejemplo), sino a través de la propia línea de comandos (ello facilitaría, por ejemplo, el uso del programa como tarea programada, ya que no debería estar presente ningún usuario para introducir los parámetros).

- Ejemplo:

```
int main(int argc, char *argv[])
{    int n;
     // ...
     // Interpretación de argumentos (si los hubiere).
     lecturaArgumentos = interpretarArgumentos(argc, argv, &n);
     if (lecturaArgumentos)
     {
          // ...
     }
}
```

# 10. COMPILACIÓN MULTIFUNCIONAL

```
// Interpretación de los posibles argumentos que pudiere haber en la invocación al programa.
// - Entradas:
// argc: Número de argumentos de entrada.
// argv: Puntero a los argumentos de entrada.
// n: Puntero a una variable en donde se dejará la longitud del vector al que se le calculará la transformada.
// - Salidas:
// 0 -> Sintaxis errónea. 1 -> Sin argumentos. 2 -> Argumento "--help". 3 -> Argumento "--n="; en este caso,
// el valor indicado se dejará sobre "n".
int interpretarArgumentos(int argc, char *argv[], int *n)
{
     int resp = (argc == 1 || argc == 2); // Si el programa no tiene argumentos (argc=1)
     if (resp && argc == 2) // o si tiene uno solo (argc=2), todo ok.
     {
          // Hay un argumento. Éste podrá ser "--help" o "--n=<tamaño_vector>".
          if (strcmp(argv[1],"--help") == 0) resp = 2;
          else
               if (strncmp(argv[1],"--n=",4) == 0)
               {
                    *n = atoi(&argv[1][4]);
                    if (*n >= 4) resp = 3;
                    else resp = 0;
               }
               else resp = 0;
     }
     return resp;
}
```

## 10. COMPILACIÓN MULTIFUNCIONAL

- Ejemplo de ejecución:

```
[aamm@localhost 17_wt_sec]$ ./wt aa bb cc
Sintaxis incorrecta. Teclee ./wt --help
[aamm@localhost 17_wt_sec]$ ./wt --help
Uso: ./wt [--n=<tamaño_vector>]
Si se indica --n=<tamaño_vector>, el tamaño ha de ser mayor o igual que 4.
Si no se indica --n=<tamaño_vector>, el tamaño será 1000000.
Ejemplo 1: ./wt
Ejemplo 2: ./wt --n=500000
[aamm@localhost 17_wt_sec]$ ./wt
Tiempo =   0.01''
[aamm@localhost 17_wt_sec]$ ./wt --n=500000
Tiempo =   0.01''
```

# 10. COMPILACIÓN MULTIFUNCIONAL

## 10.4 - MEDICIÓN DE TIEMPOS

- Fundamental medir tiempos absolutos (los relativos se obtienen con "gprof") para comparar versiones de una misma aplicación, o aplicaciones diferentes que hagan lo mismo.

- Tiempo total de un programa: Comando "time" (time <comando>).

- Tiempo de una parte de un programa: Utilizar "gettimeofday".

```
#include <sys/time.h>
...

int main(int argc, char *argv[])
{
    ...
    struct timeval iteInicio, iteFinal;
    double tiempo;
```

# 10. COMPILACIÓN MULTIFUNCIONAL

## MEDICIÓN DE TIEMPOS (continuación)

```
        // ...
        gettimeofday(&iteInicio, NULL);
        // Parte del código a medir...
        gettimeofday(&iteFinal, NULL);
        tiempo = ((double)iteFinal.tv_sec + (double)iteFinal.tv_usec/1000000) -
                 ((double)iteInicio.tv_sec + (double)iteInicio.tv_usec/1000000);
        printf("Tiempo = %6.2f''\n",tiempo);
        // ...
    }
```

# 10. COMPILACIÓN MULTIFUNCIONAL

## 10.5 - PROGRAMA DE EJEMPLO: TRANSFORMADA DE WAVELET

- Makefile:

```
CC = icc
# Compilación normal.
wt:       wt.c
          $(CC) -Wall -o wt wt.c
# Modo depuración.
debug:    wt.c
          icc -g -o wt wt.c
# Modo "profiling".
prof:     wt.c
          icc -pg -o wt wt.c
# Exhibición de resultados.
mostrar: wt.c
          icc -D MOSTRAR -Wall -o wt wt.c
# Eliminación de todo lo que no sean fuentes.
clean:
          $(RM) wt
          $(RM) *~
```

# 10. COMPILACIÓN MULTIFUNCIONAL

## PROGRAMA DE EJEMPLO: TRANSFORMADA DE WAVELET (continuación)

- Código fuente:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>

/* Transformada de Wawelet 1D con Daub-4: Versión secuencial. */

// Interpretación de los posibles argumentos que pudiere haber en la invocación
// al programa.
int interpretarArgumentos(int argc, char *argv[], int *n);

// Cálculo de la transformada.
void wt_4d(float *vector, int longVector);
```

## 10. COMPILACIÓN MULTIFUNCIONAL

```
// Función principal.
int main(int argc, char *argv[])
{
     int longVector = 1000000, n, respExe = 0, lecturaArgumentos, i;
     float *vector, parte_entera, parte_decimal;
     struct timeval iteInicio, iteFinal;
     double tiempo;
     // Interpretación de argumentos (si los hubiere).
     lecturaArgumentos = interpretarArgumentos(argc, argv, &n);
     if (lecturaArgumentos)
     {
          if (lecturaArgumentos == 2) // --help
          {
               printf("Uso: ./wt [--n=<tamaño_vector>]\n");
               printf("Si se indica --n=<tamaño_vector>, el tamaño ha de ser >= 4.\n");
               printf("Si no se indica --n=<tamaño_vector>, el tamaño será %d.\n",longVector);
               printf("Ejemplo 1: ./wt\n");
               printf("Ejemplo 2: ./wt --n=500000\n");
          }
          // La sintaxis de la llamada es correcta, y se puede calcular la transformada.
          else
          {
```

## 10. COMPILACIÓN MULTIFUNCIONAL

```
if (lecturaArgumentos == 3) longVector = n; // Tomar valor "n".
// Inicialización del vector.
vector = malloc(sizeof(float)*longVector);
for (i=0; i<longVector; i++)
{
     parte_entera = (float)(rand() % 255);
     parte_decimal = (float)(1 / ((float)(rand() % 100)));
     vector[i] = parte_entera + parte_decimal; // 0 <= vector[i] < 255
}
// ¿Mostramos vector de entrada?
#ifdef MOSTRAR
printf("ENTRADA:");
for (i=0; i<longVector; i++) printf("%8.2f ", vector[i]);
#endif
// Cálculo de la transformada.
gettimeofday(&iteInicio, NULL);
wt_4d(vector, longVector);
gettimeofday(&iteFinal, NULL);
tiempo = ((double)iteFinal.tv_sec + (double)iteFinal.tv_usec/1000000) -
((double)iteInicio.tv_sec + (double)iteInicio.tv_usec/1000000);
```

## 10. COMPILACIÓN MULTIFUNCIONAL

```
            // ¿Mostramos vector de salida?
            #ifdef MOSTRAR
            printf("\nSALIDA :"); for (i=0; i<longVector; i++) printf("%8.2f ", vector[i]);
            printf("\n");
            #endif
            // Liberación de recursos.
            free(vector);
            // Exhibición del tiempo consumido.
            printf("Tiempo = %6.2f''\n",tiempo);
        }
    }
    // Error sintáctico.
    else
    {
        printf("Sintaxis incorrecta. Teclee ./wt --help\n");
        respExe = 1;
    }
    return respExe;
}
```

## 10. COMPILACIÓN MULTIFUNCIONAL

```
// Interpretación de los posibles argumentos que pudiere haber en la invocación
// al programa.
// - Entradas:
// argc: Número de argumentos de entrada.
// argv: Puntero a los argumentos de entrada.
// n: Puntero a una variable en donde se dejará la longitud del vector al que
// se le calculará la Transformada de Wavelet.
// - Salidas:
// 0 -> Sintaxis errónea. 1 -> Sin argumentos. 2 -> Argumento "--help".
// 3 -> Argumento "--n="; en este caso, el valor indicado se dejará sobre "n".

int interpretarArgumentos(int argc, char *argv[], int *n)
{
    int resp = (argc == 1 || argc == 2); // Si el programa no tiene argumentos (argc=1)
    if (resp && argc == 2) // o si tiene uno solo (argc=2), todo ok.
    {
        // Hay un argumento. Éste podrá ser "--help" o "--n=<tamaño_vector>".
        if (strcmp(argv[1],"--help") == 0) resp = 2;
        else
        if (strncmp(argv[1],"--n=",4) == 0)
        {
            *n = atoi(&argv[1][4]);
```

## 10. COMPILACIÓN MULTIFUNCIONAL

```
                if (*n >= 4) resp = 3;
                else resp = 0;
            }
            else resp = 0;
        }
        return resp;
}

// Cálculo de la transformada.

void wt_4d(float *vector,int longVector)
{
        // Coeficientes wavelets Daub-4.
        float c0 = 0.4829629131445341;
        float c1 = 0.8365163037378079;
        float c2 = 0.2241438680420134;
        float c3 = -0.1294095225512604;
        // Otras variables.
        int i, j, mitad;
        float *tmp;
```

## 10. COMPILACIÓN MULTIFUNCIONAL

```
    // Control inicial: El vector de entrada debe contar con, al menos, 4 elementos.
    if (longVector >= 4)
    {   tmp = (float *) malloc(sizeof(float)*longVector); // Reserva del espacio necesario.
        // Bucle wavelet.
        mitad = longVector / 2;
        for (i = 0, j = 0; i < longVector - 3; i += 2, j++)
        {
            tmp[j] = c0 * vector[i] + c1 * vector[i+1] +
                     c2 * vector[i+2] + c3 * vector[i+3];
            tmp[j+mitad] = c3 * vector[i] - c2 * vector[i+1] +
                           c1 * vector[i+2] - c0 * vector[i+3];
        }
        // Ajustes finales.
        tmp[j] = c0 * vector[longVector-2] + c1 * vector[longVector-1] +
                 c2 * vector[0] + c3 * vector[1];
        tmp[j+mitad] = c0 * vector[longVector-2] - c1 * vector[longVector-1] +
                       c2 * vector[0] - c3 * vector[1];
        // Paso de los resultados, del vector temporal al vector de entrada/salida.
        for (i = 0; i < longVector; i++) vector[i] = tmp[i];
        free(tmp); // Liberación del espacio ya utilizado.
    }
}
```

# 10. COMPILACIÓN MULTIFUNCIONAL

## PROGRAMA DE EJEMPLO: TRANSFORMADA DE WAVELET (continuación)

- Compilación y ejecución:

```
[aamm@localhost 17_wt_sec]$ make mostrar
icc -D MOSTRAR -Wall -o wt wt.c
[aamm@localhost 17_wt_sec]$ ./wt --n=14
ENTRADA:  163.01   162.07    83.03   241.01   249.05   107.04 ...
SALIDA :  201.72   283.68   184.15   196.32   148.58   124.35 ...
Tiempo =   0.00''
[aamm@localhost 17_wt_sec]$ make clean
rm -f wt
rm -f *~
rm -f gmoun.out
[aamm@localhost 17_wt_sec]$ make
icc -Wall -o wt wt.c
[aamm@localhost 17_wt_sec]$ ls -l ./wt
-rwxrwxr-x 1 aamm aamm 22769 mar 28 19:09 ./wt
```

## 10. COMPILACIÓN MULTIFUNCIONAL

```
[aamm@localhost 17_wt_sec]$ time ./wt --n=50000000
Tiempo =   0.56''

real    0m4.106s
user    0m3.709s
sys     0m0.394s
[aamm@localhost 17_wt_sec]$ make clean
rm -f wt
rm -f *~
rm -f gmoun.out
[aamm@localhost 17_wt_sec]$ make -f Makefile_gcc
cc -Wall -o wt wt.c
[aamm@localhost 17_wt_sec]$ ls -l ./wt
-rwxrwxr-x 1 aamm aamm 7036 mar 28 19:09 ./wt
[aamm@localhost 17_wt_sec]$ time ./wt --n=50000000
Tiempo =   1.80''

real    0m6.796s
user    0m6.386s
sys     0m0.390s
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

## 11.1 - PROGRAMACIÓN MULTIHILO: JUSTIFICACIÓN

- La tendencia actual en el diseño de las nuevas arquitecturas de microprocesadores se centra en incrementar la eficiencia de la CPU dotando al chip de varios procesadores (también denomindos "núcleos", o "cores"), más que en seguir aumentando la frecuencia de funcionamiento del micro (este aumento de frecuencia tiene límites técnicos relacionados con el calentamiento del chip y el consumo de energía).

- Ejemplos: Intel Core Duo (arquitectura de 32 bits, con 2 núcleos), Intel Core 2 Duo (arquitectura de 64 bits, con 2 núcleos), Intel Core 2 Quad (arquitectura de 64 bits, con 4 núcleos), etc.

- Con la "colaboración" del sistema operativo, al programador (y, en alguna medida, también al compilador) le es posible particionar los algoritmos en "hilos" de ejecución paralela. Una vez arrancada una aplicación diseñada de este modo ("aplicación multihilo"), será el sistema operativo quien se ocupe de distribuir entre los distintos cores de la CPU (y de la forma más equitativa posible) los diferentes hilos componentes de la aplicación. Windows, a partir de la versión XP, y todas las distribuciones de Linux, ya son capaces de hacer esto.

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

## 11.2 - PROGRAMACIÓN MULTIHILO: EL CASO DEL INTEL C++

- A través de las directivas "OpenMP", el Intel C++ permite la programación multihilo. Requisitos:
  - Al compilar, incluir el argumento "-openmp" en el comando "icc" o "icpc" (según sea código C o C++, respectivamente).
  - En el código fuente, incluir el fichero de cabecera "omp.h".
- Método de programación:
  - Partiendo de la versión secuencial del algoritmo, detectar aquellas partes del mismo susceptibles de ser ejecutadas en paralelo.
  - Si no se detecta paralelismo, escribir el programa del modo habitual (puramente secuencial). En este caso solamente habrá un hilo.

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

## PROGRAMACIÓN MULTIHILO: EL CASO DEL INTEL C++ (continuación)

- Si, en cambio, existe paralelismo en alguna de las partes del algoritmo, acotar dicha parte y encerrarla, dentro de "#pragma omp parallel { //Código... }". Cuando la ejecución del programa llega al "#pragma omp parallel", además del hilo preexistente (que ejercerá, a partir de ahora, como hilo principal, o "maestro"), se crearán dos o más hilos adicionales, "repartiéndose" el trabajo paralelizable esos hilos, y yendo cada hilo a un core (o, si faltan cores, con reparto ecuánime). Al final de la última llave del "#pragma omp parallel", todos los hilos adicionales (o "secundarios") son destruidos y únicamente pervive el hilo maestro de la aplicación (hasta el final, o bien hasta el siguiente "#pragma omp parallel").

- Más información: Archivos "Ayuda_Openmp.pdf" y "Directivas_Openmp.pdf", en http://webs.um.es/einiesta/miwiki/doku.php?id=docencia

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

## 11.3 - PROGRAMACIÓN MULTIHILO, EJEMPLO Nº 1: TRANSFORMADA DE WAVELET

- Makefile:

```
CC = icc

# Compilación normal.
wt_multihilo:    wt_multihilo.c
                 $(CC) -openmp -Wall -o wt_multihilo wt_multihilo.c

# Exhibición de resultados.
mostrar:         wt_multihilo.c
                 icc -openmp -D MOSTRAR -Wall -o wt_multihilo wt_multihilo.c

# Eliminación de todo lo que no sean fuentes.
clean:
                 $(RM) wt_multihilo
                 $(RM) *~
                 $(RM) core*
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

## PROGRAMACIÓN MULTIHILO, EJEMPLO Nº 1: TRANSFORMADA DE WAVELET (continuación)

- Código fuente:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>
#include <omp.h>

/* Transformada de Wawelet 1D con Daub-4: Versión multihilo. */

// Interpretación de los posibles argumentos que pudiere haber en la invocación
// al programa.
int interpretarArgumentos(int argc, char *argv[], int *n);

// Cálculo de la transformada.
void wt_4d(float *vector, int longVector);
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
// Función principal.

int main(int argc, char *argv[])
{
  int longVector = 1000000, n, respExe = 0, lecturaArgumentos, i;
  float *vector, parte_entera, parte_decimal;
  struct timeval iteInicio, iteFinal;
  double tiempo;
  // Interpretación de argumentos (si los hubiere).
  lecturaArgumentos = interpretarArgumentos(argc, argv, &n);
  if (lecturaArgumentos)
  {
    if (lecturaArgumentos == 2) // --help
    {
      printf("Uso: ./wt [--n=<tamaño_vector>]\n");
      printf("Si se indica --n=<tamaño_vector>, el tamaño ha de ser mayor o igual que 4.\n");
      printf("Si no se indica --n=<tamaño_vector>, el tamaño será %d.\n",longVector);
      printf("Ejemplo 1: ./wt\n");
      printf("Ejemplo 2: ./wt --n=500000\n");
    }
    // La sintaxis de la llamada es correcta, y se puede calcular la transformada.
    else
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
{
  if (lecturaArgumentos == 3) longVector = n; // Tomar valor "n".
  // Inicialización del vector.
  vector = malloc(sizeof(float)*longVector);
  for (i=0; i<longVector; i++)
  {
    parte_entera = (float)(rand() % 255);
    parte_decimal = (float)(1 / ((float)(rand() % 100)));
    vector[i] = parte_entera + parte_decimal; // 0 <= vector[i] < 255
  }
  // ¿Mostramos vector de entrada?
  #ifdef MOSTRAR
  printf("ENTRADA:");
  for (i=0; i<longVector; i++) printf("%8.2f ", vector[i]);
  printf("\n");
  #endif
  // Cálculo de la transformada.
  gettimeofday(&iteInicio, NULL);
  wt_4d(vector, longVector);
  gettimeofday(&iteFinal, NULL);
  tiempo = ((double)iteFinal.tv_sec + (double)iteFinal.tv_usec/1000000) -
           ((double)iteInicio.tv_sec + (double)iteInicio.tv_usec/1000000);
```

## 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
      // ¿Mostramos vector de salida?
      #ifdef MOSTRAR
      printf("\nSALIDA :"); for (i=0; i<longVector; i++) printf("%8.2f ", vector[i]); printf("\n");
      #endif
      // Liberación de recursos.
      free(vector);
      // Exhibición del tiempo consumido.
      printf("Tiempo = %6.2f''\n",tiempo);
    }
  }
  // Error sintáctico.
  else
  {
    printf("Sintaxis incorrecta. Teclee ./wt --help\n");
    respExe = 1;
  }
  return respExe;
}

// Interpretación de los posibles argumentos que pudiere haber en la invocación
// al programa.
// - Entradas:
//   argc: Número de argumentos de entrada.
//   argv: Puntero a los argumentos de entrada.
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
//   n: Puntero a una variable en donde se dejará la longitud del vector al que
//   se le calculará la Transformada de Wavelet.
// - Salidas:
//   0 -> Sintaxis errónea. 1 -> Sin argumentos. 2 -> Argumento "--help".
//   3 -> Argumento "--n="; en este caso, el valor indicado se dejará sobre "n".

int interpretarArgumentos(int argc, char *argv[], int *n)
{
  int resp = (argc == 1 || argc == 2); // Si el programa no tiene argumentos (argc=1)
  if (resp && argc == 2)             // o si tiene uno solo (argc=2), todo ok.
  {
    // Hay un argumento. Éste podrá ser "--help" o "--n=<tamaño_vector>".
    if (strcmp(argv[1],"--help") == 0) resp = 2;
    else
      if (strncmp(argv[1],"--n=",4) == 0)
      {
        *n = atoi(&argv[1][4]);
        if (*n >= 4) resp = 3;
        else resp = 0;
      }
      else resp = 0;
  }
  return resp;
}
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
// Cálculo de la transformada.

void wt_4d(float *vector,int longVector)
{
  // Coeficientes wavelets Daub-4.
  float c0 = 0.4829629131445341;float c1 = 0.8365163037378079;
  float c2 = 0.2241438680420134;float c3 = -0.1294095225512604;
  // Otras variables.
  int i, j, mitad;
  float *tmp;
  // Control inicial: El vector de entrada debe contar con, al menos, 4 elementos.
  if (longVector >= 4)
  {
    tmp = (float *) malloc(sizeof(float)*longVector); // Reserva del espacio necesario.
    // Bucle wavelet.
    mitad = longVector / 2;
    // Fragmento de código paralelizado.
    #pragma omp parallel private(i,j)  // Las variables "i" y "j" serán locales para cada hilo.
    {
      #pragma omp single // Esto sólo lo ejecutará uno de los hilos
      {
        printf("Nº hilos en ejecución: %d\n",omp_get_num_threads());
        #ifdef MOSTRAR printf("Parejas (hilo,i): "); #endif
      }
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
    #pragma omp for nowait // Cada hilo se apropia de una pasada. La cláusula "nowait" hace que
    // cuando un hilo se percata de que ha de salir del bucle, no espere a que los demás hilos
    // abandonen también el bucle. Sólo puede haber una variable de control para el bucle. Además,
    // como cada hilo toma una "i" local distinta, la "j" hay que recalcularla en cada pasada.
    for (i = 0; i < longVector - 3; i += 2)
    {
      #ifdef MOSTRAR printf("(%d,%d) ",omp_get_thread_num(),i); #endif
      j = i / 2;
      tmp[j] =       c0 * vector[i] + c1 * vector[i+1] + c2 * vector[i+2] + c3 * vector[i+3];
      tmp[j+mitad] = c3 * vector[i] - c2 * vector[i+1] + c1 * vector[i+2] - c0 * vector[i+3];
    }
    // Punto de sincronización. Necesario debido a la cláusula "nowait" del "for" anterior.
    #pragma omp barrier
    // Ajustes finales.
    #pragma omp single
    {
      tmp[mitad-1]     = c0*vector[longVector-2]+c1*vector[longVector-1]+c2*vector[0]+c3*vector[1];
      tmp[longVector-1]= c0*vector[longVector-2]-c1*vector[longVector-1]+c2*vector[0]-c3*vector[1];
    }
  } // Fin ámbito #pragma omp parallel private(i,j)
  // Paso de los resultados, del vector temporal al vector de entrada/salida.
  for (i = 0; i < longVector; i++) vector[i] = tmp[i];
  free(tmp); // Liberación del espacio ya utilizado.
} }
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

## PROGRAMACIÓN MULTIHILO, EJEMPLO Nº 1: TRANSFORMADA DE WAVELET (continuación)

- Compilación, ejecución y comparación con la versión secuencial:

```
[aamm@localhost 19_wt_multihilo]$ make mostrar
icc -openmp -D MOSTRAR -Wall -o wt_multihilo wt_multihilo.c
[aamm@localhost 19_wt_multihilo]$ ./wt_multihilo --n=14
ENTRADA:   163.01    162.07     83.03    241.01    249.05    107.04     20.02    233.04     45.04
142.03    86.01    87.03   167.03   212.04
Nº hilos en ejecución: 2
Parejas (hilo,i): (0,0) (1,6) (0,2) (1,8) (0,4) (1,10)
SALIDA :   201.72    283.68    184.15    196.32    148.58    124.35    273.61   -104.37     91.87
-152.03   -85.74    -7.74     6.68   -39.20
Tiempo =   0.02''
[aamm@localhost 19_wt_multihilo]$ make clean
rm -f wt_multihilo
rm -f *~
rm -f core*
[aamm@localhost 19_wt_multihilo]$ make
icc -openmp -Wall -o wt_multihilo wt_multihilo.c
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
[aamm@localhost 19_wt_multihilo]$ time ./wt_multihilo --n=50000000
Nº hilos en ejecución: 2
Tiempo =   0.38''

real    0m3.897s
user    0m3.795s
sys     0m0.454s
[aamm@localhost 19_wt_multihilo]$ time ../17_wt_sec/wt --n=50000000
Tiempo =   0.48''

real    0m4.286s
user    0m3.886s
sys     0m0.339s
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

## 11.4 - PROGRAMACIÓN MULTIHILO, EJEMPLO Nº 2: SUMAS SENOIDALES

- Makefile:

```
CC = icc

# Modo secuencial.
sumasenos:      sumasenos.c
                $(CC) -Wall -o sumasenos_secuencial sumasenos.c

# Modo multihilo.
multihilo:      sumasenos.c
                icc -openmp -D MULTIHILO -Wall -o sumasenos_multihilo sumasenos.c

# Eliminación de todo lo que no sean fuentes.
clean:
                $(RM) sumasenos_secuencial
                $(RM) sumasenos_multihilo
                $(RM) *~
                $(RM) gmoun.out
                $(RM) core*
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

## PROGRAMACIÓN MULTIHILO, EJEMPLO Nº 2: SUMAS SENOIDALES (continuación)

- Código fuente:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>

#ifdef MULTIHILO
#include <omp.h>
#endif

/* Suma de senos.

  - Objetivo: Evaluar la mejora de rendimiento que supone el uso de la programación multihilo.

  - Posibles directivas de compilación:
    • MULTIHILO: Generación de un programa que construya varios hilos.

  - Sintaxis: sumasenos [--help] [--n=<nº_enteros_aleatorios>] [--n_hilos=<nº_hilos>] [--mostrar] */
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
// Interpretación de los posibles argumentos que pudiere haber en la invocación al programa.
int interpretarArgumentos(int argc, char *argv[], int *n, int *n_hilos, int *mostrar);

// Cálculo de la suma de "n" senos.
double calcularSuma(int n, int n_hilos, int mostrar);

// Función principal.

int main(int argc, char *argv[])
{
  int lecturaArgumentos, n = 1000000, n_hilos = 0, mostrar = 0;
  int respExe = 0;
  double suma;
  struct timeval iteInicio, iteFinal;
  double tiempo;
  lecturaArgumentos = interpretarArgumentos(argc, argv, &n, &n_hilos, &mostrar);
  if (lecturaArgumentos)
  {
    if (lecturaArgumentos == 2) // --help
    {
      printf("Uso: ./sumasenos [--n=<nº_enteros_aleatorios>] [--n_hilos=<º_hilos>] [--mostrar]\n");
      printf("Si no se indica --n=<número_enteros_aleatorios>, su valor será %d.\n",n);
      printf("Si no se indica --n_hilos=<número_hilos>, habrá tantos hilos como cores.\n");
      printf("Si no se indica --mostrar, su valor será 0 (FALSE).\n"); }
```

## 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
    // La sintaxis de la llamada es correcta, y se puede calcular la suma.
    else
    {
      #ifndef MULTIHILO
      n_hilos = 0;
      #endif
      // Cálculo de la suma.
      gettimeofday(&iteInicio, NULL);
      suma = calcularSuma(n, n_hilos, mostrar);
      gettimeofday(&iteFinal, NULL);
      tiempo = ((double)iteFinal.tv_sec + (double)iteFinal.tv_usec/1000000) -
               ((double)iteInicio.tv_sec + (double)iteInicio.tv_usec/1000000);
      printf("Suma   = %6.2f (%d sumandos).\n", suma, n);
      // Exhibición del tiempo consumido.
      printf("Tiempo = %6.2f''\n",tiempo);
    }
  }
  else
  {
    printf("Sintaxis incorrecta. Teclee ./sumasenos --help\n");
    respExe = 1;
  }
  return respExe;
}
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
// Interpretación de los posibles argumentos que pudiere haber en la invocación
// al programa.
//
// - Entradas:
//   argc:    Número de argumentos de entrada.
//   argv:    Puntero a los argumentos de entrada.
//   n:       Puntero a una variable en donde se dejará el número de sumandos.
//   n_hilos: Puntero al número de hilos a crear (sólo con directiva MULTIHILO).
//   mostrar: Puntero a una bandera booleana que indica si deben mostrarse los
//            sumandos por pantalla.
// - Salidas:
//   0 -> Sintaxis errónea.
//   1 -> Sin argumentos.
//   2 -> Argumento "--help".

int interpretarArgumentos(int argc, char *argv[],
                          int *n, int *n_hilos, int *mostrar)
{
  int resp = argc <= 4;
  int i, paramOk = 0;
  if (argc > 1) // Existe, al menos, un argumento.
  {
```

## 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
    if (strcmp(argv[1],"--help") == 0) resp = 2;
    else
    {
      for (i = 1; i < argc && resp; i++)
      {
        if (strncmp(argv[i],"--n=",4) == 0)
        {
          paramOk = 1; *n = atoi(&argv[i][4]); if (*n <= 0) resp = 0;
        }
        if (strncmp(argv[i],"--n_hilos=",10) == 0)
        {
          paramOk = 1; *n_hilos = atoi(&argv[i][10]); if (*n_hilos < 0) resp = 0;
        }
        if (strncmp(argv[i],"--mostrar",9) == 0)
        {
          paramOk = 1; *mostrar = 1;
        }
      }
      if (!paramOk) resp = 0;
    }
  }
  return resp;
}
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
// Cálculo de la suma de "n" senos.
//
// - Entradas:
//   n:       Número de sumandos.
//   n_hilos: Número de hilos a crear (sólo con directiva MULTIHILO).
//   mostrar: Bandera booleana que indica si deben mostrarse los sumandos por pantalla.
// - Salidas: Suma total de los "n" valores.

double calcularSuma(int n, int n_hilos, int mostrar)
{
  double suma = 0, sumando, suma_parcial = 0;
  int i, n_medios = n / 2;
  #ifdef MULTIHILO
  // Establecimiento del número de hilos. Casuística:
  // Si n_hilos = 0, no se fuerza un número concreto de número de hilos (se deja al compilador que
  // lo decida, en función del número de cores de la máquina).
  // Si n_hilos > 0, se establece manualmente ese número de hilos, pero en caso de ser superior a 2,
  // no tendrá efecto sobre el rendimiento, ya que en este ejemplo sólo trabajaremos con 2 secciones.
  if (n_hilos > 0) omp_set_num_threads(n_hilos);
  #pragma omp parallel
  {
    #pragma omp single // Esto lo hace sólo un hilo.
    {
      if (n_hilos == 0) n_hilos = omp_get_num_threads();
    }
  }
  #endif
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
// SUMA ESTRICTAMENTE SECUENCIAL.

#ifndef MULTIHILO
if (mostrar) printf("SUMANDOS (hilo,sumando): ");
for (i = 1; i <= n; i++)
{
  sumando = sin(i);
  if (mostrar) printf("(%d,%5.2f) ", 1, sumando);
  suma = suma + sumando;
}
if (mostrar) printf("\n");
#endif

// SUMA MULTIHILO

#ifdef MULTIHILO
if (mostrar) printf("SUMANDOS (hilo,sumando): ");
#pragma omp parallel shared(suma) private(i,sumando,suma_parcial)
// Variables compartidas por los dos hilos: "suma".
// Variables locales de cada hilo: "i", "sumando" y "suma_parcial".
{
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
  #pragma omp sections // Definición de las dos secciones (dos hilos, por tanto).
  {
    #pragma omp section // Hilo nº 1.
    {
      for (i = 1; i <= n_medios; i++)
      {
        sumando = sin(i); if (mostrar) printf("(%d,%5.2f) ", omp_get_thread_num(), sumando);
        suma_parcial = suma_parcial + sumando;
      }
    }
    #pragma omp section // Hilo nº 2.
    {
      for (i = n_medios + 1; i <= n; i++)
      {
        sumando = sin(i); if (mostrar) printf("(%d,%5.2f) ", omp_get_thread_num(), sumando);
        suma_parcial = suma_parcial + sumando;
      }
    }
  }
  // Aquí no se llega hasta que no hayan concluido los dos hilos. Sección crítica: Actualizamos "suma".
  #pragma omp critical
  {
    suma = suma + suma_parcial;
  }
  #pragma omp barrier // Sincronización de hilos.
}
if (mostrar) printf("\n");
#endif
return suma; }
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

## PROGRAMACIÓN MULTIHILO, EJEMPLO Nº 2: SUMAS SENOIDALES (continuación)

- Compilación y ejecución:

```
[aamm@localhost 18_suma_senos]$ make
icc -Wall -o sumasenos_secuencial sumasenos.c
[aamm@localhost 18_suma_senos]$ make multihilo
icc -openmp -D MULTIHILO -Wall -o sumasenos_multihilo sumasenos.c
sumasenos.c(30): remark #1419: external declaration in primary source file
[aamm@localhost 18_suma_senos]$ ls ./sumasenos_secuencial ./sumasenos_multihilo
./sumasenos_multihilo  ./sumasenos_secuencial
[aamm@localhost 18_suma_senos]$ ./sumasenos_secuencial --n=10 --mostrar
SUMANDOS (hilo,sumando): (1, 0.84) (1, 0.91) (1, 0.14) (1,-0.76) (1,-0.96) (1,-0.28) (1,
0.66) (1, 0.99) (1, 0.41) (1,-0.54)
Suma   =   1.41 (10 sumandos).
Tiempo =   0.00''
[aamm@localhost 18_suma_senos]$ ./sumasenos_multihilo --n=10 --mostrar
SUMANDOS (hilo,sumando): (0, 0.84) (0, 0.91) (0, 0.14) (0,-0.76) (0,-0.96) (1,-0.28) (1,
0.66) (1, 0.99) (1, 0.41) (1,-0.54)
Suma   =   1.41 (10 sumandos).
Tiempo =   0.02''
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
[aamm@localhost 18_suma_senos]$ ./sumasenos_secuencial --n=100000000
Suma    =    1.71 (100000000 sumandos).
Tiempo =    7.29''
[aamm@localhost 18_suma_senos]$ ./sumasenos_multihilo --n=100000000
Suma    =    1.71 (100000000 sumandos).
Tiempo =    3.92''
[aamm@localhost 18_suma_senos]$ ./sumasenos_multihilo --n=100000000 --n_hilos=10
Suma    =    1.71 (100000000 sumandos).
Tiempo =    4.60''
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

## 11.5 - USO EFICIENTE DE LA MEMORIA

- Un aprovechamiento apropiado de las memorias RAM y caché es fundamental para optimizar el rendimiento de las aplicaciones, sobre todo si éstas son de cálculo intensivo.

- El acceso a la RAM no supone trasladar entre ésta y la CPU el contenido de una única dirección física, sino el de un bloque completo de varios KB, que es alojado en la memoria caché, mucho más rápida que la RAM. Que el programador colabore para que los accesos posteriores (al menos los más cercanos en el tiempo) sean a la caché y no a la RAM puede mejorar enormemente la eficiencia de la aplicación.

- Esta colaboración consiste en proporcionar la mayor localidad espacial posible en los accesos a memoria.

- Ejemplo nº 1: Matriz (bidimensional, por tanto) implementada como un array (espacio reservado con `malloc` y liberado con `free`) de "m" filas por "n" columnas, donde las "n" columnas de la fila "i" están justo delante de las "n" columnas de la fila "i+1", con $0 < i < m$. En este contexto, un recorrido de la matriz fila a fila resulta mucho más eficiente que un recorrido columna a columna.

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

## USO EFICIENTE DE LA MEMORIA (continuación)

- Ejemplo nº 2: Elección correcta de los tipos de datos, teniendo en cuenta el dominio de los valores almacenables, el espacio requerido para codificar dicho dominio y la precisión exigida. De este modo, utilizar el tipo `double` (IEEE 754 de doble precisión, 64 bits) para, por ejemplo, almacenar valores enteros que van a estar entre -30000 y 30000 (cuando podíamos emplear para tal fin el tipo `short int` -16 bits en complemento a 2-) supone:

  - Un desperdicio de memoria, y lo que es peor, una ralentización de los accesos a la misma, ya que hay que trasladar entre la RAM y la caché muchos más bytes para acceder al mismo número de datos.
  - Un enlentecimiento de los cálculos, si los hubiere, ya que la aritmética en punto flotante es más compleja que la aritmética en punto fijo.

Es importante, no obstante, valorar también la precisión requerida para los valores a almacenar. Por ejemplo, si hemos de guardar en un array números reales, la elección natural sería, por las razones expuestas, el tipo `float` (IEEE 754 de simple precisión, 32 bits), en detrimento de `double`, pero siempre y cuando no tuviésemos que codificar valores muy cercanos al 0 o valores muy altos. Téngase en cuenta que con los float disponemos de menos bits para la mantisa, y esto puede implicar, en el primer caso, redondear a 0 el valor (nefasto, sobre todo si la variable asociada es el factor de un producto) o, en el segundo, ajustar a un número demasiado alejado del valor real que se pretende codificar.

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

## 11.6 - USO EFICIENTE DE LA MEMORIA: PROGRAMA DE EJEMPLO (RECORRIDO MATRICIAL)

- Makefile:

```
memoria:        memoria.c
                icc -Wall -o memoria_int memoria.c

doubles:        memoria.c
                icc -D DOUBLE -Wall -o memoria_doubles memoria.c

clean:
                $(RM) memoria_*
                $(RM) memoria*~
                $(RM) core*
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

## USO EFICIENTE DE LA MEMORIA: PROGRAMA DE EJEMPLO (RECORRIDO MATRICIAL)

- Código fuente:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

/* Recorrido de matrices (instanciación de sus elementos, generados de modo aletorio).

   Objetivos: 1. Demostrar la importancia que para el rendimiento tiene un uso óptimo de la memoria.
              2. Saber determinar qué tipo de dato es el más adecuado para cada variable, conociendo
                 el rango de representación necesario y los efectos que sobre el acceso a memoria
                 tiene el tamaño de dicho tipo de dato (uso eficiente de la caché).

   Posibles directivas de compilación:
     • DOUBLE: El tipo de datos al que pertenecerán los elementos de la matriz.

   Sintaxis: memoria [--help] [--n=<orden_matriz>] [--col] [--mostrar] */
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
#define ORDEN 1000

// Interpretación de los posibles argumentos que pudiere haber en la invocación al programa.
int interpretarArgumentos(int argc, char *argv[], int *n, int *recorrerPorFilas, int *mostrar);

// Instanciación de los elementos de una matriz cuadrada de orden "n". El recorrido podrá hacerse
// fila a fila, o columna a columna (se pretende destacar el peso que esta decisión tiene en el
// rendimiento del proceso).
#ifdef DOUBLE
void rellenarMatriz(double **matriz, int n, int recorrerPorFilas, int mostrar);
#else
void rellenarMatriz(short int **matriz, int n, int recorrerPorFilas, int mostrar);
#endif

// Función principal.
int main(int argc, char *argv[])
{
  int lecturaArgumentos, n = ORDEN, recorrerPorFilas = 1, mostrar = 0;
  int respExe = 0, i;
  #ifdef DOUBLE
  double **matriz;
  #else
  short int **matriz;
  #endif
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
struct timeval iteInicio, iteFinal;
double tiempo;
lecturaArgumentos = interpretarArgumentos(argc, argv, &n, &recorrerPorFilas, &mostrar);
if (lecturaArgumentos)
{
  if (lecturaArgumentos == 2) // --help
  {
    printf("Uso: ./memoria [--n=<orden_matriz>] [--col] [--mostrar]\n");
    printf("Si no se indica --n=<orden_matriz>, su valor será %d.\n",ORDEN);
    printf("Si no se indica --col, la matriz será recorrida columna a columna.\n");
    printf("Si no se indica --mostrar, su valor será 0 (FALSE).\n");
  }
  // La sintaxis de la llamada es correcta, y se puede rellenar la matriz.
  else
  {
    // Reserva de espacio. "matriz" es un puntero a otro puntero. Este último puntero, a su vez, es
    // el primero de una lista de "n" punteros que apuntan al primer double/short int de cada fila
    // de la matriz (el siguiente malloc reserva espacio para esta lista de punteros a filas, pero
    // aún  no hemos reservado el espacio para las filas).
    #ifdef DOUBLE
    matriz = (double **) malloc(sizeof(double *) * n);
    #else
    matriz = (short int **) malloc(sizeof(short int *) * n);
    #endif
```

## 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
      // Reservamos el espacio para las filas.
      for (i = 0; i < n; i++)
        #ifdef DOUBLE
        matriz[i] = (double *) malloc(sizeof(double) * n);
        #else
        matriz[i] = (short int *) malloc(sizeof(short int) * n);
        #endif
      // Instanciación de los elementos de la matriz.
      gettimeofday(&iteInicio, NULL);
      rellenarMatriz(matriz, n, recorrerPorFilas, mostrar);
      gettimeofday(&iteFinal, NULL);
      tiempo = ((double)iteFinal.tv_sec + (double)iteFinal.tv_usec/1000000) -
               ((double)iteInicio.tv_sec + (double)iteInicio.tv_usec/1000000);
      // Exhibición del tiempo consumido.
      printf("Tiempo = %10.2f''\n",tiempo);
      // Liberación de memoria.
      for (i=0; i < n; i++) free(matriz[i]); // Liberamos, una a una, cada fila.
      free(matriz); // Liberamos los punteros a las filas.
    }
  }
  else
  { printf("Sintaxis incorrecta. Teclee ./memoria --help\n");
    respExe = 1; }
  return respExe; }
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
// Interpretación de los posibles argumentos que pudiere haber en la invocación
// al programa.
//
// - Entradas:
//   argc:             Número de argumentos de entrada.
//   argv:             Puntero a los argumentos de entrada.
//   n:                Puntero a una variable en donde se dejará el orden de la matriz.
//   recorrerPorFilas: Puntero a una bandera booleana que indica si la instanciación de
//                     elementos debe hacerse fila a fila (TRUE) o columna a columna (FALSE).
//   mostrar:          Puntero a una bandera booleana que indica si deben mostrarse la
//                     matriz por pantalla.
// - Salidas:
//   0 -> Sintaxis errónea. 1 -> Sin argumentos. 2 -> Argumento "--help".

int interpretarArgumentos(int argc, char *argv[],
                          int *n, int *recorrerPorFilas, int *mostrar)
{
  int resp = argc <= 4;
  int i, paramOk = 0;
  if (argc > 1) // Existe, al menos, un argumento.
  {
    if (strcmp(argv[1],"--help") == 0) resp = 2;
    else
```

## 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
    {
      *recorrerPorFilas = 1;
      for (i = 1; i < argc && resp; i++)
      {
        if (strncmp(argv[i],"--n=",4) == 0)
        {
          paramOk = 1;
          *n = atoi(&argv[i][4]); if (*n <= 0) resp = 0;
        }
        if (strncmp(argv[i],"--col",5) == 0)
        {
          paramOk = 1;
          *recorrerPorFilas = 0;
        }
        if (strncmp(argv[i],"--mostrar",9) == 0)
        {
          paramOk = 1; *mostrar = 1;
        }
      }
      if (!paramOk) resp = 0;
    }
  }
  return resp;
}
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
// Instanciación de los elementos de una matriz cuadrada de orden "n". El recorrido podrá ser fila a
// fila, o columna a columna (se pretende destacar el peso de esta decisión sobre el rendimiento).
//
// - Entradas:
//   matriz:           Puntero a la matriz a rellenar.
//   n:                Orden de la matriz (cuadrada).
//   recorrerPorFilas: Si vale 1, la instanciación de los elementos se hará según un recorrido fila a
//                     fila (el más eficiente, teniendo en cuenta cómo se acomodan los elementos de
//                     la matriz en la RAM, y el funcionamiento del sistema de memoria caché); si
//                     vale 0, el recorrido se llevará a cabo columna a columna.
// - Salidas:          No hay.

#ifdef DOUBLE
void rellenarMatriz(double **matriz, int n, int recorrerPorFilas, int mostrar)
#else
void rellenarMatriz(short int **matriz, int n, int recorrerPorFilas, int mostrar)
#endif
{
  int i, j;
  #ifdef DOUBLE
  srand48(time(NULL));
  #else
  srand(time(NULL));
  #endif
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
  for (i = 0; i < n; i++)  // filas
  {
    for (j = 0; j < n; j++) // columnas
    {
      #ifdef DOUBLE
      if (recorrerPorFilas) matriz[i][j] = drand48() * 10;
      else matriz[j][i] = drand48() * 10;
      if (mostrar) printf("%4.2f ",matriz[i][j]);
      #else
      if (recorrerPorFilas) matriz[i][j] = rand() % 10;
      else matriz[j][i] = rand() % 10;
      if (mostrar) printf("%2d ",matriz[i][j]);
      #endif
    }
    if (mostrar) printf("\n");
  }
}
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

## USO EFICIENTE DE LA MEMORIA: PROGRAMA DE EJEMPLO (RECORRIDO MATRICIAL)

- Compilación y tests de ejecución:

```
[aamm@localhost 21_memoria]$ make
icc -Wall -o memoria_int memoria.c
[aamm@localhost 21_memoria]$ make doubles
icc -D DOUBLE -Wall -o memoria_doubles memoria.c
[aamm@localhost 21_memoria]$ ls ./memoria_int ./memoria_doubles
./memoria_doubles  ./memoria_int
[aamm@localhost 21_memoria]$ ./memoria_int --n=3 --mostrar
 3  5  6
 4  8  0
 6  8  4
Tiempo =       0.00''
[aamm@localhost 21_memoria]$ ./memoria_doubles --n=3 --mostrar
2.43 3.59 1.97
0.28 2.43 6.60
6.37 8.82 0.63
Tiempo =       0.00''
```

## 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

```
[aamm@localhost 21_memoria]$ ./memoria_int --n=12000
Tiempo =       4.46''
[aamm@localhost 21_memoria]$ ./memoria_int --n=12000 --col
Tiempo =       5.83''
[aamm@localhost 21_memoria]$ ./memoria_doubles --n=12000
Tiempo =       8.27''
[aamm@localhost 21_memoria]$ ./memoria_doubles --n=12000 --col
Tiempo =      12.79''
```

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

## 11.7 - RECOMENDACIONES VARIAS

1. En las instrucciones de asignación, las conversiones de tipos, tanto si éstas son implícitas como si son explícitas ("`cast`"), deben evitarse siempre que ello sea posible, sobre todo si se encuentran dentro de bucles de "larga duración".

2. Otras conversiones aparentemente simples e inocuas, como los truncamientos ("`trunc`"), los redondeos ("`round`") o los valores absolutos ("`abs`") también hay que intentar evitarlas.

3. Cuando existan varias alternativas válidas a la hora de elegir el tipo de dato de una variable, optar siempre por la más simple (`short int` en lugar de `int`, `float` en vez de `double`, etc). Ello suele redundar en un consumo menor de memoria, con los beneficios que esto procura para la caché (ya comentado en el apartado anterior) y, cuando haya cálculos de por medio, en un menor tiempo de cómputo (la aritmética en punto fijo, por ejemplo, es más rápida que la aritmética en punto flotante).

# 11. TÉCNICAS DE OPTIMIZACIÓN BASADAS EN LA ARQUITECTURA DEL COMPUTADOR

## RECOMENDACIONES VARIAS (continuación)

4. Probar las opciones de optimización ofrecidas por el compilador. Su omisión, en la línea de comando de `icc` o `icpc`, no suele perjudicar significativamente el nivel de eficiencia de la aplicación. Estas opciones son las siguientes:

   - `-O0`: Deshabilita todas las optimizaciones. Es la opción que implícitamente adoptan `icc` e `icpc` cuando se incluye el argumento `-g` (opción de depuración).
   - `-O1`: Habilita la mayoría de las optimizaciones, excepto aquellas que provocan un aumento considerable del tamaño del código generado (hay casos en los que un ejecutable grande puede tener repercusiones muy negativas en su rendimiento).
   - `-O2`: Habilita la práctica totalidad de las optimizaciones orientadas a maximizar la velocidad de ejecución de la aplicación. Si se omite, es la opción que "de oficio" adoptan `icc` e `icpc`.
   - `-O3`: Habilita todas las optimizaciones de `-O2` y además añade algunas otras relacionadas con una compilación más eficiente del código de los bucles. No en todos los casos construye programas que sean más rápidos que los obtenidos con `-O2`.

# 12. DETERMINACIÓN DE LAS CARACTERÍSTICAS DE UNA CPU INTEL

## 12.1 - INSTRUCCIÓN CPUID

- Las CPUs Intel incluyen, dentro de su repertorio de instrucciones, una instrucción que permite identificar sus características en tiempo de ejecución. Esta instrucción se denomina "`cpuid`".

- `cpuid` admite como parámetro de entrada un código, denominado "función CPUID", que indica el tipo de información que pretendemos obtener. Este código debe colocarse en el registro EAX antes de hacer la invocación a `cpuid`. La información devuelta es colocada, en formato normalizado, dentro de los registros EBX y/o ECX y/o EDX.

- Algunas de las funciones CPUID (en constante crecimiento, conforme van fabricándose nuevos modelos de CPU) son las siguientes:

  - Función 0: Nombre del fabricante del chip.
  - Función 1: Familia de micros, modelo de CPU, extensiones multimedia disponibles, etc.
  - Función 2: Descriptores de la memoria caché (tamaño del TLB, número de vías, etc).

# 12. DETERMINACIÓN DE LAS CARACTERÍSTICAS DE UNA CPU INTEL

## INSTRUCCIÓN CPUID (continuación)

- Función 3: Número de serie del chip.
- Función 11: Topología del procesador.
- Etc.

- La determinación de qué instrucciones SIMD (flujo simple de instrucciones, múltiple de datos) son admitidas por una CPU se obtiene con la función CPUID 1, a través de los bits siguientes:

  - Soporta instrucciones MMX: Registro EDX, bit 23 (númeración de los bits: $b_{31}$, $b_{30}$, ..., $b_1$, $b_0$).
  - Soporta instrucciones SSE: Registro EDX, bit 25.
  - Soporta instrucciones SSE2: Registro EDX, bit 26.
  - Soporta instrucciones SSE3: Registro ECX, bit 0.
  - Soporta instrucciones SSSE3: Registro ECX, bit 9.
  - Soporta instrucciones SSS4.1: Registro ECX, bit 19.
  - Soporta instrucciones SSS4.2: Registro ECX, bit 20.

# 12. DETERMINACIÓN DE LAS CARACTERÍSTICAS DE UNA CPU INTEL

## 12.2 - DETECCIÓN DE LAS EXTENSIONES MULTIMEDIA DISPONIBLES

- Código fuente:

```
#include <stdio.h>

#define TAMFAB 12

/* Obtención de datos de la CPU relevantes para instrucciones SIMD (uso de la instrucción CPUID).
   Mas información en    http://www.intel.com/Assets/PDF/appnote/241618.pdf
   Arquitecturas Multimedia, 31/03/10. */

typedef struct {
  int MMX;
  int SSE;
  int SSE2;
  int SSE3;
  int SSSE3;
  int SSE41;
  int SSE42; } infoSIMD;
```

# 12. DETERMINACIÓN DE LAS CARACTERÍSTICAS DE UNA CPU INTEL

```
// Obtención del fabricante de la CPU.
void obtener_fabricante(char *fabricante);

// Obtención de datos relacionados con el soporte de instrucciones SIMD.
void obtener_SIMD(infoSIMD *datosSIMD);

int main(void)
{
  char fabricante[TAMFAB+1];
  infoSIMD datosSIMD;
  // Fabricante.
  obtener_fabricante(fabricante);
  printf("Fabricante: %s\n", fabricante);
  // Soporte SIMD.
  obtener_SIMD(&datosSIMD);
  if (datosSIMD.MMX) printf("MMX       : Sí\n"); else printf("MMX       : No\n");
  if (datosSIMD.SSE) printf("SSE       : Sí\n"); else printf("SSE       : No\n");
  if (datosSIMD.SSE2) printf("SSE2      : Sí\n"); else printf("SSE2      : No\n");
  if (datosSIMD.SSE3) printf("SSE3      : Sí\n"); else printf("SSE3      : No\n");
  if (datosSIMD.SSSE3) printf("SSSE3     : Sí\n"); else printf("SSSE3     : No\n");
  if (datosSIMD.SSE41) printf("SSE41     : Sí\n"); else printf("SSE41     : No\n");
  if (datosSIMD.SSE42) printf("SSE42     : Sí\n"); else printf("SSE42     : No\n");
  return 0;
}
```

# 12. DETERMINACIÓN DE LAS CARACTERÍSTICAS DE UNA CPU INTEL

```
// Obtención del fabricante de la CPU.
// Entradas: Puntero a una cadena en donde se dejará el identificador del fabricante de la CPU.
// Salidas:  No hay.

void obtener_fabricante(char *fabricante)
{
  int reg_ebx, reg_edx, reg_ecx, aux;
  char c, resto;
  // Función 0 de CPUID: Obtención del fabricante. Los 12 caracteres se dejan en EBX, EDX y ECX.
  __asm__ volatile (
    "movl $0,%%eax\n\t"
    "cpuid\n\t"
    "movl %%ebx,%0\n\t"
    "movl %%edx,%1\n\t"
    "movl %%ecx,%2\n\t"
    : "=m" (reg_ebx), "=m" (reg_edx), "=m" (reg_ecx)
    : "m" (reg_ebx), "m" (reg_edx), "m" (reg_ecx)
    : "memory"
  );
  // Extracción del resultado.
  for (c = 0; c < TAMFAB; c++)
  {
    if (c < 4) aux = reg_ebx; else if (c < 8) aux = reg_edx; else aux = reg_ecx;
    resto = c % 4;
    fabricante[c] = (aux >> (resto*8)) & 0xFF;
  }
  fabricante[TAMFAB] = '\0';
}
```

# 12. DETERMINACIÓN DE LAS CARACTERÍSTICAS DE UNA CPU INTEL

```
// Obtención de datos relacionados con el soporte de instrucciones SIMD.
// Entradas: Puntero a una estructura "infoSIMD".
// Salidas:  No hay.

void obtener_SIMD(infoSIMD *datosSIMD)
{
  int reg_edx, reg_ecx;
  // Función 1 de CPUID: Obtención del tipo de CPU, modelo, soporte SIMD, etc.
  __asm__ volatile (
    "movl $1,%%eax\n\t"
    "cpuid\n\t"
    "movl %%edx,%0\n\t"
    "movl %%ecx,%1\n\t"
    : "=m" (reg_edx), "=m" (reg_ecx)
    : "m" (reg_edx), "m" (reg_ecx)
    : "memory"
  );
  // Extracción de resultados.
  datosSIMD->MMX   = reg_edx & 0x00800000;
  datosSIMD->SSE   = reg_edx & 0x02000000;
  datosSIMD->SSE2  = reg_edx & 0x04000000;
  datosSIMD->SSE3  = reg_ecx & 0x00000001;
  datosSIMD->SSSE3 = reg_ecx & 0x00000200;
  datosSIMD->SSE41 = reg_ecx & 0x00080000;
  datosSIMD->SSE42 = reg_ecx & 0x00100000;
}
```

# 12. DETERMINACIÓN DE LAS CARACTERÍSTICAS DE UNA CPU INTEL

## DETECCIÓN DE LAS EXTENSIONES MULTIMEDIA DISPONIBLES (continuación)

- Prueba en una CPU Intel Core 2 Duo:

```
[aamm@localhost 20_cpuid]$ cat /proc/cpuinfo | grep -i "model name"
model name      : Intel(R) Core(TM)2 Duo CPU     T5670  @ 1.80GHz
model name      : Intel(R) Core(TM)2 Duo CPU     T5670  @ 1.80GHz
[aamm@localhost 20_cpuid]$ ./datos_cpuid
Fabricante: GenuineIntel
MMX       : Sí
SSE       : Sí
SSE2      : Sí
SSE3      : Sí
SSSE3     : Sí
SSE41     : No
SSE42     : No
```

## BIBLIOGRAFÍA

1. Bernabé, G. "Arquitecturas Multimedia, material didáctico". Departamento de Ingeniería y Tecnología de Computadores (Universidad de Murcia). Tema 2, 2010.

2. Mittal, M., Pelegy, A. y Weiser, A. "MMX Technology Architecture Overview". Intel Corporation. Intel Technology Journal Q3, 1997.

3. Thakkar, S. y Huff, T. "The Internet Streaming SIMD Extensions". Intel Corporation. Intel Technology Journal Q2, 1999.

4. Wolf III., J. "Programming Methods for the Pentium III Processor's Streaming SIMD Extensions Using the Vtune Performance Enhancement Environment". Intel Corporation. Intel Technology Journal Q2, 1999.

5. Intel. "Intel C++ Compiler for Linux Reference, document 307777-002US". Intel Corporation. 2006.

6. Intel. "Intel Processor Identification and the CPUID Instruction, application note 485". Intel Corporation. 2009.