PRÁCTICA 1: DESARROLLO DE CÓDIGO PARA EL PROCESADOR ARM

INTRODUCCIÓN

Este documento es el guion de la primera práctica de la asignatura Proyecto Hardware del Grado en Ingeniería Informática de Escuela de Ingeniería y Arquitectura la Universidad de Zaragoza.

En esta primera práctica vamos a optimizar el rendimiento de un juego acelerando las funciones computacionalmente más costosas. A partir del código facilitado en C, se desarrollará código para un procesador ARM y se ejecutará sobre un emulador de un procesador ARM. Para ello, tendremos que trabajar con el entorno de desarrollo Keil µVision IDE. El emulador de este entorno también simula el procesador lo que permite estimar el tiempo de ejecución del mismo.

También se deberá entender la funcionalidad del código suministrado y realizar una versión en ensamblador ARM optimizando las funciones críticas. Se medirá el rendimiento respecto al código optimizado generado por el compilador y se documentarán los resultados.

Se medirá tamaño en bytes, número de instrucciones ejecutadas y tiempo de ejecución. Para medir el tiempo, se programarán los temporizadores internos de la placa. Se verificará que todas las versiones del código den un resultado equivalente.

ÍNDICE

OBJETIVOS

- Interactuar con un microcrontrolador y ser capaces de **ejecutar** y **depurar** .
- Profundizar en la interacción C / Ensamblador.
- Ser capaces de depurar el **código ensamblador** que genera un compilador a partir de un lenguaje en alto nivel.
- Conocer la estructura segmentada en tres etapas del procesador ARM 7, uno de los más utilizados actualmente en sistemas empotrados.
- Familiarizarse con el entorno Keil µVision sobre Windows, con la generación cruzada de código para ARM y con su depuración.
- Aprender a analizar el rendimiento y la estructura de un programa.
- Desarrollar código en ensamblador ARM: adecuado para optimizar el rendimiento.
- **Optimizar** código: tanto en ensamblador ARM, como utilizando las opciones de optimización del compilador.
- Entender la finalidad y el funcionamiento de las ***Aplication Binary Interface***, ABI, en este caso el estándar **ATPCS** (*ARM-Thumb Application Procedure Call Standard*), y combinar de manera eficiente código en ensamblador con código en C.
- Saber **depurar** código siguiendo el estado arquitectónico de la máquina: contenido de los registros y de la memoria.
- Comprobar automáticamente que varias implementaciones de una función mantienen la misma funcionalidad.

CONOCIMIENTOS PREVIOS NECESARIOS

En esta asignatura cada estudiante necesitará aplicar contenidos adquiridos en asignaturas previas, en particular, Arquitectura y Organización de Computadores I y II.

ENTORNO DE TRABAJO

Esta práctica se realiza con **Keil µVision**, el mismo entorno que ya se utilizó en primero para AOC1. Trabajaremos sobre el microcontrolador LPC2105 (el mismo que se usaba en primero).

El entorno µVision es capaz de editar, depurar, compilar código en C y ensamblador, además de analizar el rendimiento de los binarios generados. En prácticas sucesivas veremos como también es capaz de emular diferentes periféricos y entrada salida del microcontrolador.

Se utilizará la versión 4 o 5. En Moodle de la asignatura tenéis información del proceso de instalación y fuentes. También podéis registraros e instalar el software desde la web de Keil.

MATERIAL ADICIONAL

En el curso Moodle de la asignatura puede encontrarse el siguiente material para prácticas:

- Manuales de Keil µVision
- Documentación del sistema emulado
- Manuales de la arquitectura de referencia:
  - Breve resumen del repertorio de instrucciones ARM.
  - Manual de la arquitectura ARM.
  - Información sobre el ABI de ARM: ATPCS
  - Documento con información sobre variables en C
- Directrices para redactar una memoria técnica, imprescindible para redactar la memoria de la práctica. Es donde se define la estructura de la memoria a entregar.

Y para la realización de la práctica 1 además: proyecto para Keil µVision con los códigos fuentes del juego, este documento, las diapositivas de la presentación y un video explicativo.

ESTRUCTURA DE LA PRÁCTICA

La práctica completa se debe realizar en 3 sesiones y será la base para la práctica siguiente, la 4ª semana tendrá lugar la entrega presencial del trabajo y una semana después la memoria a través de Moodle. Las fechas concretas se anunciarán en el sitio web de la asignatura (Moodle).

El trabajo de la asignatura se realiza online, tanto dentro del horario asignado como fuera del mismo. Las tutorías se atenderán en el horario asignado o fijado como tutorías.

Antes de la primera sesión es imprescindible haberse leído este guion y conocer la documentación suministrada. Para entrar en la segunda sesión de prácticas es necesario traer el código fuente de los apartados 5 y 6 lo más avanzados posible. La tercera sesión se debería dedicar a realizar las métricas de los apartados 7 y 8, por lo que se deberá mostrar antes de empezar el correcto funcionamiento de todo lo anterior.

EL JUEGO: REVERSI

El **Reversi** (también conocido como Othello) es uno de los juegos de tablero más populares en el mundo. La clave de su éxito es que las reglas son muy sencillas, aunque jugar bien es muy difícil.

Cierta empresa quiere lanzar un sistema que juegue al Reversi contra una persona y que se ejecutará en un procesador ARM7. Como primer paso han escrito una versión beta del programa en C (como versión beta está sujeta a fallos, avisad si veis alguno). Pero

no están contentos con el tiempo de cálculo del código y os piden que lo reduzcáis. Para ello os proponen acelerar la función más crítica del código: **`patron_volteo()`**.

TAREAS A REALIZAR

En esta primera práctica nos centraremos en la implementación eficiente de las que en principio se consideran las funciones más críticas del juego. Para ello primero buscamos entender el funcionamiento del sistema y el comportamiento de las funciones resaltadas. Posteriormente evaluaremos el rendimiento y estudiaremos las opciones de optimización del compilador para mejorar las prestaciones del código generado.

Por tanto, tenéis que:

Paso 1: Estudiar la documentación.

Paso 2: Estudiar y depurar el código inicial del juego.

Estudiar el código inicial en C y la especificación de las funciones **`patron_volteo()`** y **`ficha_valida()`** -esta última es invocada desde `patron_volteo`()-.

a) Insertar en el proyecto ejemplo anterior el fichero de código del juego. Creando las cabeceras necesarias. Modificar la función **`main()`** para que tras inicializar el sistema se llame a la función **`reversi8()`**. Las funciones no llamadas desde el **`main()`** no deben ser visibles.

b) Debéis monitorizar la **ejecución de ambas funciones en C, verificando** sobre el tablero en memoria **la correcta ejecución**.

c) Revisa la información facilitada por el compilador, si hay errores o avisos, entiende que ocurre y corrígelo. En particular, deberás comprobar los tipos utilizados para definir las distintas variables.

d) Observar el código en ensamblador generado por el compilador y depurarlo paso a paso, prestar atención a los cambios en memoria y en los registros del procesador.

e) Dibuja el mapa de memoria (código, variables globales, pila, etc).

f) Dibuja el marco de activación en pila y el paso de parámetros de las llamadas a funciones.

g) Analizar la utilización de variables tipo char. En caso de posibles problemas sustituirlas por otros tipos mejor definidos.

Paso 3: Analizar el rendimiento. Hacer uso del analizador de rendimiento (***profiling***) para analizar el árbol de ejecución y los tiempos de ejecución de las diferentes funciones del juego. Determinar las funciones principales del juego y las de mayor peso de cómputo. En la memoria deberéis reflejar los resultados obtenidos.

Paso 4: Realizar `patron_volteo_arm_c()` en ensamblador ARM. Realiza una función equivalente a **`patron_volteo()`** en ensamblador ARM, aplicando las optimizaciones que consideres oportuno, pero manteniendo la llamada a la función en C **`ficha_valida()`** original.

El código ARM debe tener la misma estructura que el código C: cada función, cada variable o cada condición que exista en el código C debe poder identificarse con facilidad en la versión de ensamblador. Para ello se deberán incluir los comentarios oportunos.

IMPORTANTE: **DEBE MANTENERSE LA MODULARIDAD DEL CÓDIGO**. Es decir no se puede eliminar la llamada a la función **`ficha_valida()`** e integrarla dentro de la función **`patron_volteo_arm_c()`**. No se puede eliminar la llamada, o eliminar el paso de parámetros. Tampoco se puede acceder a las variables locales de una función desde otra. Esto último es un fallo muy grave que implica un suspenso en la práctica.

Cuando hagáis la llamada en ensamblador debe ser una llamada convencional a una función. Es decir, hay que pasar los parámetros, utilizando el estándar **ATPCS** a través de los registros correspondientes. No sirve hacer un salto sin pasar parámetros.

El marco de pila (o bloque de activación) de las distintas combinaciones debe ser idéntico al creado por el compilador para que la comparación sea justa. **La descripción del marco de pila utilizado deberá aparecer convenientemente explicado en la memoria.**

En todos los casos se debe garantizar que una función no altere el contenido de los registros privados de las otras funciones.

Paso 5: Realizar una nueva función `patron_volteo_arm_arm()`. La función **`ficha_valida`** ha sido creada por el programador como función independiente por claridad y legibilidad del código. Sin embargo, solo se le llama desde **`patron_volteo`**, por ello se puede eliminar dicha llamada incrustando (inlining) el código de la misma en vuestra función en ARM. De esa forma nos evitamos el llamar a la subrutina y las sobrecargas asociadas a esta operación.

Crea una nueva función **`patron_volteo_arm_arm()`** con esta configuración.

IMPORTANTE: **DEBE MANTENERSE EL ALGORITMO DEL CÓDIGO**. Es decir, no se puede modificar en las versiones ARM el algoritmo realizado en C. En caso de querer hacer pequeñas modificaciones en el algoritmo es mejor consultarlo primero con el profesorado.

Paso 6: Verificación automática y comparación de resultados. Los procesos de verificación y optimización son una parte fundamental del desarrollo del software y deben tenerse en cuenta desde el principio del tiempo de vida del software.

Tenemos diversas implementaciones de **`patron_volteo`** (C-C, C-ARM-C, C-ARM). Como durante la fase de desarrollo del código en ARM es muy fácil que se cometa algún error **no** fácilmente **detectable al depurar** a mano. Vuestro código debe **verificar que las distintas combinaciones generan la misma salida.** Este proceso lo tendréis que realizar múltiples veces, por lo que **se debe automatizar**.

Para ello, debéis crear una nueva función **`patrón_volteo_test`** con las mismas entradas y salidas que **`patrón_volteo`** original. Dentro de esta función se llamará a las diferentes versiones (**`patron_volteo`**, **`patron_volteo_arm_c`** y **`patron_volteo_arm_arm`**)

comprobando que el resultado coincide. Si no coinciden, la función indicará el error, se quedará en un bucle infinito y no retornará.

No tiene sentido probar cada vez todas las posibles combinaciones de entradas. Sin embargo, sí que es deseable realizar las comprobaciones en un conjunto pequeño, pero representativo de todos los posibles casos. Diseñar un conjunto o **vector de pruebas** que cubra los principales escenarios. Deberéis razonar los experimentos.

El día de la corrección se debe mostrar el funcionamiento de la verificación automática.

Paso 7: Medidas de rendimiento. Una vez comprobado que vuestro código funciona bien y que el entorno mide bien los tiempos, vamos a medir los tiempos de ejecución sobre el procesador.

Estamos interesados en medir las funciones críticas (no en todo el programa completo). Debéis comparar el **tamaño del código** en **bytes** de cada versión de las funciones realizadas (C original, ARM, etc). También queremos **evaluar el tiempo de ejecución** de cada una de estas configuraciones al realizar el cálculo de ficha negra en fila 2 - columna 3. Calcular las métricas de las combinaciones y **razonar los resultados obtenidos.**

Los valores del tamaño de código y los tiempos de ejecución debéis presentarlos en la entrega de la práctica **el día de la corrección**.

Paso 8: Optimizaciones del compilador. Por defecto el compilador de C genera un código seguro y fácilmente depurable (*debug*[1]) facilitando la ejecución paso a paso en alto nivel. Esta primera versión es buena para depurar, pero es muy ineficiente. Los compiladores disponen de diversas opciones de compilación o *flags* que permiten configurar la generación de código. En concreto hay un conjunto de opciones que permiten aplicar heurísticas de optimización de código buscando mejorar la velocidad o el tamaño. Estas configuraciones se suelen especificar con los *flags* `-O0 (por defecto, sin optimizar), -O1, -O2 (código en producción), -O3, -Os`.

Estudiar el impacto en el rendimiento del código C de los niveles de optimización (`-O0, -O1, -O2, -O3, -Os`) y compararlo con las otras versiones en rendimiento, tamaño y número de instrucciones ejecutadas. Como los cambios en el nivel de optimización solo afectan al código de alto nivel hecho en C, sólo es obligatorio hacer el estudio para la combinación C-C de funciones **`patron_volteo()`** y **`ficha_valida()`**. Siendo opcional en la combinación **`patron_volteo_arm_c`** ya que os puede servir para verificar que vuestro código sigue correctamente el ATPCS.

Compararlo con los resultados anteriores y razonar la respuesta. Para ello, todos estos resultados deben estar disponibles **el día de la corrección** y quedar claramente reflejados y comentados **en la memoria**.

[1] Te sorprendería saber la cantidad de software comercial generado sin optimizar y en modo debug.

¿CÓMO FUNCIONA EL CÓDIGO FACILITADO?

El código que nos dan permite jugar a una versión reducida del Reversi en un tablero de 8x8.

Las casillas centrales empiezan con una diagonal ocupada por dos piezas negras y la otra por dos piezas blancas. El procesador jugará con las blancas y el jugador usará las negras. El primer movimiento lo hacen las negras (funciona al revés que el ajedrez).

Las reglas de Reversi son sencillas:

- Cada jugador coloca sus piezas turnándose de una en una.
- Sólo puedes colocar tu pieza en una posición si al hacerlo rodeas a una o varias fichas del rival en alguna dirección (norte-sur, este-oeste, o alguna de las dos diagonales).
- Las fichas rodeadas cambian de color. En la figura 1 se puede ver el tablero inicial, los posibles movimientos que tiene el jugador negro (en gris), y el tablero resultante tras mover el jugador negro a la fila 1, columna 2.
- Si no hay ninguna posición libre tienes que pasar.
- Cuando ningún jugador pueda mover se acaba el juego. El que tenga más fichas de su color gana la partida.

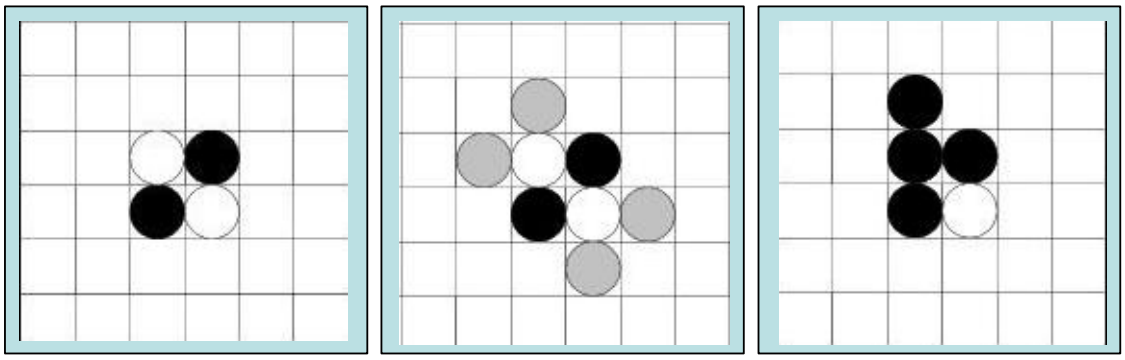


Figura 1: (a) Tablero inicial **(b) Posibles movimientos** **(c) Tablero tras mover en 1-2**

Como en esta versión todavía no hay ningún dispositivo de entrada/salida, el jugador debe realizar sus movimientos escribiendo directamente en memoria. El funcionamiento es el siguiente:

1. Pon un **`breakpoint`** en la función reversi8 en la línea en la que se invoca a la función **`esperar_mov(ready)`**. Con esto se consigue que cada vez que le toque mover al jugador el simulador pare y nos deje introducir nuestro movimiento. Después de esto puedes darle a ejecutar.

2. Mira en tu código dónde está en memoria el *array* **`tablero`** y pon un monitor de memoria en esa dirección. Te da varias opciones para representar los datos de la memoria; por defecto se muestran en hexadecimal. Configúralo tal y como aparece en la figura 2, como el tablero es de 8x8, es conveniente mostrar 8

columnas. Si la dirección no es múltiplo de 8 el tablero no se verá bien. Se ha solucionado con una directiva de alineamiento. De esta forma el tablero es visible para el jugador. En el tablero cada byte representa una posición: las posiciones con 0 están vacías, las que tienen un 1 contienen fichas blancas, y las que tienen un 2 contienen fichas negras (ver figura 2).

3. Justo después de tablero hay tres posiciones de memoria que se corresponden con las variables tipo **`char`**: **`fila`**, **`columna`** y **`ready`**. Fila y columna se usan para que el jugador especifique su movimiento escribiendo directamente en memoria, y **`ready`** para avisar al procesador de que ya se ha movido. Para mover, coloca el cursor en la posición de memoria en la que quieras escribir y teclea el valor. Después de hacer tu movimiento dale otra vez a ejecutar. Tras varios segundos verás como el tablero se actualiza con tu movimiento y también el del rival. Sigue jugando hasta que termine la partida. En la figura 3 se puede ver cómo se visualiza el tablero inicial, cómo se realiza un movimiento y cómo queda tras el movimiento.

4. Si no puedes mover, introduce como movimiento fila 8 columna 8. Como las filas y columnas van numeradas de 0 a 7, el procesador comprenderá que no puedes mover.

5. El código no comprueba que tus movimientos sean válidos. De hecho, escribiendo en memoria podéis hacer tantas trampas como queráis por ejemplo cambiando el tablero. Esto puede ser útil para depurar ya que podéis escribir el tablero que os interese sin tener que jugar hasta llegar a él. Los movimientos del procesador sí que deben ser correctos en todo momento.

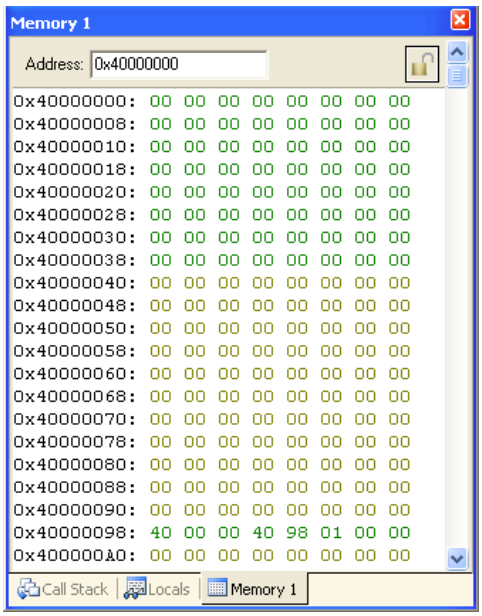


Figura 2: configurando el tablero en Keil, debes ajustar las dimensiones de la ventana a partir de la dirección de inicio

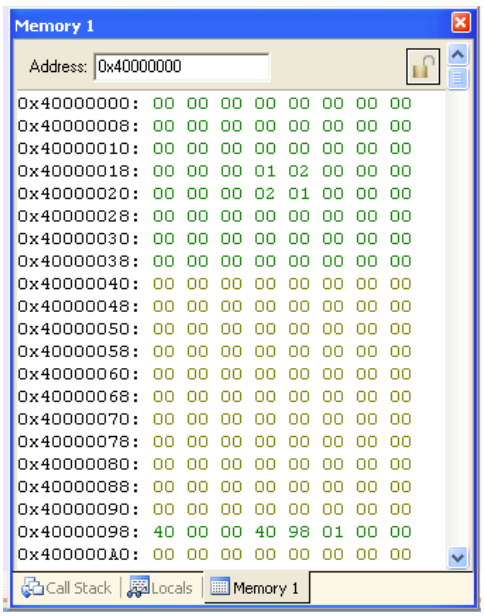
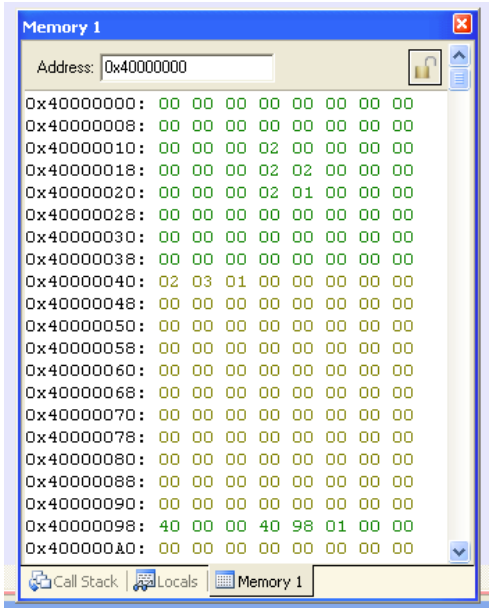


Figura 3: (a) tablero inicial tal y como debe verse tras ejecutarse "init_table"



(b) El tablero se actualiza con el movimiento indicado y la respuesta del programa

ALGUNOS CONSEJOS DE PROGRAMACIÓN EFICIENTE PARA OPTIMIZAR EL CÓDIGO ENSAMBLADOR

A la hora de evaluar este trabajo se valorará que cada estudiante haya sido capaz de optimizar el código ARM. **Cuanto más rápido** y pequeño sea el código (y menos instrucciones se ejecuten), **mejor será la valoración de la práctica**. También se valorará que se reduzcan los accesos a memoria.

Algunas ideas que se pueden aplicar son:

- No comencéis a escribir el código en ensamblador hasta tener claro cómo va a funcionar. Si diseñáis bien el código a priori, el número de instrucciones será mucho menor que si lo vais escribiendo sobre la marcha.
- Optimizar el uso de los registros. Las instrucciones del ARM trabajan principalmente con registros. Para operar con un dato de memoria debemos cargarlo en un registro, operar y, por último, y sólo si es necesario, volverlo a guardar en memoria. Mantener en los registros algunas variables que se están utilizando frecuentemente permite ahorrar varias instrucciones de lectura y escritura en memoria. Cuando comencéis a pasar del código C original a ensamblador debéis decidir qué variables se van a guardar en registros, tratando de minimizar las transferencias de datos con memoria.
- Utilizar instrucciones de transferencia de datos múltiples como LDMIA, STMIA, PUSH o POP que permiten que una única instrucción mueva varios datos entre la memoria y los registros.
- Utilizar instrucciones con ejecución condicional, también llamadas instrucciones predicadas. En el repertorio ARM gran parte de las instrucciones pueden predicarse. Por ejemplo, el siguiente código:

```
if (a == 2) { b++ }
else { b = b - 2 }
```

Con instrucciones predicadas sería:

```
CMP   r0, #2            #compara con 2
ADDEQ r1,r1,#1          #suma si r0 es 2
SUBNE r1,r1,#2          #resta si r0 no es 2
```

Mientras que sin predicados sería:

```
CMP   r0, #2      #compara con 2
BNE   resta       # si r0 no es 2 saltamos a la resta
ADD   r1,r1,#1    #suma 1
B     cont        #continuamos la ejecución sin restar
Resta: SUBNE      r1,r1,#2    #resta 2
Cont:
```

Hay otros ejemplos útiles en las transparencias de la práctica.

- Utilizar instrucciones que realicen más de una operación. Por ejemplo la instrucción MLA r2,r3,r4,r5 realiza la siguiente operación: r2 = r3*r4 + r5.
- Utilizar las opciones de desplazamiento para multiplicar. Las operaciones de multiplicación son más lentas (introducen varios ciclos de retardo). Para multiplicar/dividir por una potencia de dos basta con realizar un desplazamiento que además puede ir integrado en otra instrucción. Por ejemplo:
    - A= B + 2*C puede hacerse sencillamente con ADD R1, R2, R3, lsl #1
    - A= B + 10*C puede hacerse sencillamente con ADD R1, R2, R3, lsl #3 y ADD R1, R1, R3, lsl #1 (A = B + 8*C + 2*C)

Sacar partido de los modos de direccionamiento registro base + offset en los cálculos de la dirección en las instrucciones load/store. Por ejemplo, para acceder a A[4] podemos hacer LDR R1, [R2, #4] (si es un array de elementos de tipo char) o LDR R1, [R2, #16] (si es un array de enteros).

NOTA: como el código a desarrollar es pequeño es probable que alguna de estas optimizaciones no sea aplicable a vuestro código, pero muchas sí que lo serán y debéis utilizarlas.

APARTADO OPCIONAL 1:

Hemos estudiado el impacto en el rendimiento del código C de las opciones compilación con optimización (-O1, -O2…). La gran ventaja de que sea el compilador el que aplique las optimizaciones es que nos permite mantener claridad y portabilidad

en el código fuente, mientras obtenemos rendimiento en el ejecutable. Si hay algún cambio, lo único que deberemos hacer es recompilar para sacar de nuevo partido.

Conforme el compilador aplica las heurísticas el código se hace menos claro. Mira el manual y el código en ensamblador generado por el compilador. Explica qué técnicas ha aplicado en cada versión y como las ha empleado.

Busca optimizar tus funciones ARM aplicando las mismas técnicas. ¿Eres capaz de mejorar el rendimiento obtenido por el compilador?

En algunos casos, nosotros tenemos información sobre el algoritmo o la implementación que el compilador desconoce o debe ser conservador. ¿Eres capaz de optimizar las funciones en C para ayudar al compilador a generar "mejor" código?

EVALUACIÓN DE LA PRÁCTICA

La práctica se presentará online **aproximadamente** el 16 de octubre.

En esta sesión se entregarán los códigos fuentes y se comprobará que funciona, que al compilar no aparecen *warnings* y que el código es correcto, así como que el trabajo presentado cumple los requisitos de este documento.

La memoria habrá que presentarla **aproximadamente** el 23 de octubre. Se entregará en Moodle.

Las fechas y horarios definitivos se publicarán en la página web de la asignatura (*moodle*).

ANEXO 1: REALIZACIÓN DE LA MEMORIA

La memoria de la práctica tiene diseño libre, la estructura de la memoria se define en el documento Ayuda_elaboracion_memoria_tecnica.pdf y es obligatorio que incluya los siguientes contenidos:

1. Resumen ejecutivo (una cara como máximo). El resumen ejecutivo es un documento independiente del resto de la memoria que describe brevemente qué habéis hecho, por qué lo habéis hecho, qué resultados obtenéis y cuáles son vuestras conclusiones.
2. Código fuente comentado. Además, cada función debe incluir una cabecera en la que se explique cómo funciona, qué parámetros recibe y dónde los recibe y para qué usa cada registro (por ejemplo, en el registro 4 se guarda el puntero a la primera matriz…).
3. Mapa de memoria
4. Descripción de las optimizaciones realizadas al código ensamblador.
5. Resultados de la comparación entre las distintas versiones de las funciones.
6. Análisis de rendimiento y comparativa de las distintas versiones y opciones de compilación (tiempo de ejecución, tamaño de código, etc.)
7. Descripción de los problemas encontrados en la realización de la práctica y sus soluciones.
8. Conclusiones

Se valorará que el texto sea **claro y conciso**. Cuánto más fácil sea entender el funcionamiento del código y vuestro trabajo, mejor. **Utilizad el documento de recomendaciones para la redacción de una memoria técnica** disponible en la página web de la asignatura (moodle).

ANEXO 2: ENTREGA DE LA MEMORIA

La entrega de la memoria será a través de la página web de la asignatura (*moodle* en http://add.unizar.es). Debéis enviar un fichero comprimido en formato ZIP con los siguientes documentos:

1. Memoria en formato PDF.
2. Código fuente de los apartados A y B en formato texto.

Se debe mandar un único fichero por pareja. El fichero se nombrará de la siguiente manera:

p1_NIP-Apellidos_Estudiante1_NIP-Apellidos_Estudiante2.zip

Por ejemplo: p1_345456-Gracia_Esteban_45632-Arribas_Murillo.zip

Los ficheros independientes deben también identificar los nombres de la pareja.