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 comparará el rendimiento respecto al código optimizado generado por el compilador y se documentarán los resultados. Se medirá tamaño en bytes y tiempo de ejecución y se verificará que todas las versiones del código den un resultado equivalente.

ÍNDICE

INTRODUCCIÓN ---- 1
ÍNDICE ---- 1
OBJETIVOS ---- 2
CONOCIMIENTOS PREVIOS NECESARIOS ---- 2
ENTORNO DE TRABAJO ---- 2
MATERIAL ADICIONAL ---- 3
ESTRUCTURA DE LA PRÁCTICA ---- 3
EL JUEGO: SUDOKU ---- 3
TAREAS A REALIZAR ---- 5
¿CÓMO FUNCIONA EL CÓDIGO FACILITADO? ---- 8
APARTADO OPCIONAL 1: ---- 11
APARTADO OPCIONAL 2: ---- 11
EVALUACIÓN DE LA PRÁCTICA ---- 11
ANEXO 1: REALIZACIÓN DE LA MEMORIA ---- 12
ANEXO 2: ENTREGA DE LA MEMORIA ---- 12

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 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: SUDOKU

El Sudoku (en japonés: 数独, sūdoku) es uno de los juegos matemáticos más populares en el mundo. La clave de su éxito es que las reglas son muy sencillas, aunque resolverlo puede llegar a ser muy difícil.

El sudoku se presenta normalmente como un tablero o **cuadrícula** de 9 × 9, compuesta por sub-cuadrículas de 3 × 3 denominadas recuadros o **regiones**. Algunas **celdas** ya contienen números, conocidos como números dados (o a veces **pistas**).

El objetivo es rellenar las celdas vacías, con un número en cada una de ellas, de tal forma que cada columna, fila y región contenga los números 1–9 solo una vez.

Las reglas son sencillas:

- Se parte de una cuadrícula donde ciertas celdas ya contienen una pista
- Para completar el sudoku se deben rellenar las celdas vacías de forma que cada fila, cada columna y cada región contenga todos los números del 1 al 9.
- Cada número de la solución aparece solo una vez en cada una de las tres "direcciones", de ahí el "los números deben estar solos" que evoca el nombre del juego.

Los métodos de resolución son múltiples. La mayoría de sudokus de nivel medio o difícil son complejos de resolver y requieren técnicas sofisticadas que llevan la cuenta de los valores o **candidatos** posibles para cada celda. Se trata de reducir el número de candidatos hasta encontrar una celda con un solo candidato. A partir de esta nueva información, se repite el proceso de reducir candidatos hasta encontrar la solución final.

Examinemos los posibles candidatos de la celda R6-C4 (fila 6, columna 4) de la figura 1. Mirando la fila 6 observamos que los valores 4, 3 y 7 ya están en uso, como solamente pueden aparecer una vez por fila, podemos eliminarlos de la lista de candidatos. Inspeccionando la columna podemos eliminar además el 9, 8 y 5. Centrándonos en la región vemos que tampoco pueden ser ni el 1 ni el 6. ¿Qué valores permanecen en la lista de candidatos? Únicamente el 2.

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	5			3					
R2					9				5
R3		9	6	7		5		3	
R4		8		9			6		
R5			5	8	6	1	4		
R6			4			3		7	
R7		7		5		9	2	6	
R8	6				8				
R9						2			1

Figura 1. Ejemplo de resolución de Sudoku.

Normalmente este procedimiento se realiza escribiendo los posibles candidatos con un lápiz sobre el papel, y conforme se desarrolla el juego se van tachando los que ya no son candidatos (ver Figura 2).

Una vez asignado en la celda R6-C4 el valor 2, podemos tacharlo en todas las listas de candidatos de las celdas de la fila 6, la columna 4 y de la región central (ver Figura 2).

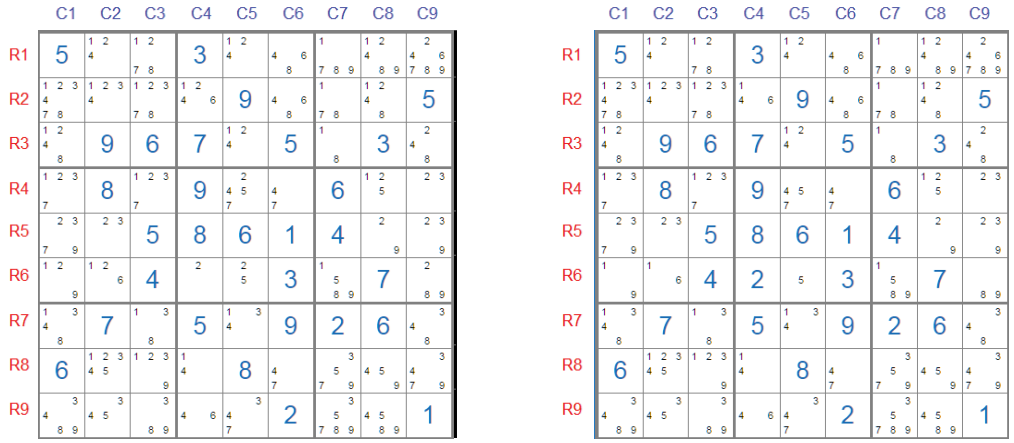

Figura 2. Actualización de candidatos tras inserción de valor.

Cierta empresa quiere lanzar un sistema, que se ejecutará en un procesador ARM7, para facilitar que una persona resuelva el rompecabezas. PH_sudoku permitirá al jugador/a ahorrar este rutinario trabajo manteniendo los candidatos de cada celda, permitiendo la edición y rellenado de celdas con nuevos valores y la correspondiente actualización de la cuadrícula. Esto facilitará la resolución de sudokus.

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 crear y acelerar la función más crítica del código: **`candidatos_actualizar_c()`**.

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 **`candidatos_actualizar_c()`** y **`candidatos_propagar_c()`**-esta última es invocada desde **`candidatos_actualizar_c()`**-. En el código inicial en C os encontraréis con el pseudocódigo del comportamiento de la función **`candidatos_actualizar_c()`**. Debéis **implementarla en C verificando** sobre la cuadrícula en memoria **la correcta ejecución**, con la función **`cuadricula_candidatos_verificar()`** debéis comprobar que el cálculo de los candidatos de todo la cuadricula ha sido realizado satisfactoriamente.

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 **`sudoku9x9()`**. 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.

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: Implementar la función `candidatos_actualizar_arm_c()` en ensamblador ARM. Realiza una función equivalente a **`candidatos_actualizar_c()`** en ensamblador ARM, aplicando las optimizaciones que consideres oportuno, pero manteniendo la llamada a la función en C **`candidatos_propagar_c()`** original. También deberás escribir el código ARM de **candidatos_propagar_arm()** que podrá ser invocada por la función en C **candidatos_actualizar_c_arm()**.

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 **`candidatos_propagar_c()`** e integrarla dentro de la función **`candidatos_actualizar_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 `candidatos_actualizar_arm_arm()` que fusione las 2 funciones en ensamblador realizadas en el apartado anterior. La función **`candidatos_propagar_c`** ha sido creada por el programador como función independiente por claridad y legibilidad del código. Sin embargo, solo se le llama desde **`candidatos_actualizar`**, 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 **`candidatos_actualizar_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 **`candidatos_actualizar`** (C-C, ARM-C, C-ARM, ARM-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, la función **sudoku9x9** realiza esta tarea invocando a las diferentes versiones sobre cuadriculas diferentes. Dentro de esta función se llama a las diferentes versiones comprobando que el resultado coincide con la solución.

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.

Nos interesa 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. Calcular las métricas de las combinaciones, comparar 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

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

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) y -O3`.

Estudiar el impacto en el rendimiento del código C de los niveles de optimización (`-O0, -O1, -O2, -O3`) y compararlo con las otras versiones en rendimiento, tamaño y número de instrucciones ejecutadas. Los cambios en el nivel de optimización sólo deberían afectar al rendimiento del código de alto nivel escrito en C y la funcionalidad no debería cambiar. Sin embargo, a veces aparecen errores. Estos errores no son culpa del compilador, son errores que ya estaban en el código pero que no eran visibles. Los errores más típicos son no seguir correctamente el AATPCS, o no asignar ***volatile*** a una variable que cambia externamente (esto os pasará más adelante). La opción -Otime, Optimize for time en los menús de Keil, permite realizar optimizaciones adicionales que pueden incrementar el tamaño de los binarios como loop unrolling o inlining. Para ver estos efectos debéis por favor activar "Optimize for time" al activar el nivel de optimización O3.

Todos estos resultados deben estar disponibles **el día de la corrección** y quedar claramente reflejados y comentados **en la memoria**. Nota: no ganar al compilador en `-O3` no es un fracaso. Los compiladores son muy buenos optimizando código. El objetivo no es tanto que ganéis al compilador como que conozcáis estos flags de optimización y los uséis en el futuro.

¿CÓMO FUNCIONA EL CÓDIGO FACILITADO?

La información del tablero está guardada en una estructura de datos denominada `cuadricula`, definidas en `tableros.h`, consistente en una matriz de celdas. Se han creado diversas cuadriculas para poder depurar y verificar con la solución el correcto funcionamiento de las distintas implementaciones.

Los datos necesarios para la celda están codificados en 16 bits (ver Figura 3). Los 4 bits de menor peso contienen el valor actual en binario (0…9). El siguiente bit indica si se trata de una pista inicial (1) o una celda inicialmente vacía (0). El siguiente bit se utilizará más adelante para marcar si se ha detectado un error en esa celda. De los 10 bits restantes, los nueve bits de mayor peso se reservan para ir almacenando durante la ejecución la lista de los posibles candidatos (mediante un mapa de bits). El bit 15 se usa para saber si el 9 es un candidato, mientras que el bit 7 se usa para el 1. El bit 6 no se utiliza actualmente. Los candidatos se codifican con lógica negada. El valor 0 indica TRUE, mientras que el valor 1 indica FALSE.

Candidatos										E	P	Valor			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figura 3. Codificación de campos en 16 bits.

EJEMPLO:
0x9a10 = 100111000 0 0 1 0000

100111000 indica que los candidatos son: 8, 7, 3, 2, y 1. El siguiente bit no se usa. El siguiente indica que no se ha detectado un error. El siguiente indica que el valor de la celda es uno de los valores iniciales. Finalmente 0101 indica que la celda está vacía.

Para simplificar el cálculo de la dirección efectiva de las celdas en matrices en C, se ha añadido un **padding** haciendo que el ancho de la cuadricula sea potencia de 2. Esto hace que las cuadriculas ocupen más espacio en memoria, pero simplifica la gestión de direcciones y podría acelerar el código.

Busca en tu código la dirección de memoria inicial del *array* `cuadricula` (por ejemplo, puedes ir ejecutando paso a paso y mirando los valores de los registros o poner un **`breakpoint`** en la línea de código de interés) y pon un monitor de memoria en esa dirección (menú *view / memory Windows*). 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 4, como la cuadricula es de 9x16 por el padding, es conveniente mostrar 16 columnas en el formato "`unsigned short`" disponible con el ratón botón derecho sobre la memoria. Si la dirección asignada por el enlazador no es múltiplo de 16 la cuadricula no se verá bien. Se puede solucionar con una directiva de alineamiento (`.align`) en `tablero.h`. De esta forma el tablero es visible para el jugador humano.

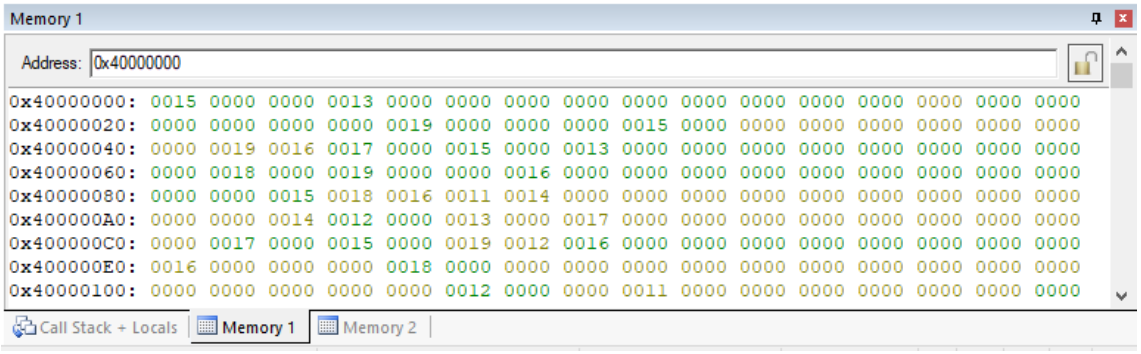

Figura 4. Cuadrícula inicial tal y como debe verse

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 8 bits) o LDR R1, [R2, #16] (si es un array de elementos de 32 bits). Además, en ARM se puede desplazar uno de los operandos de la dirección para hacer multiplicaciones de potencias de 2.

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?

APARTADO OPCIONAL 2:

Como ya se ha comentado las cuadriculas en memoria se guardan añadiendo un padding para hacer el ancho potencia de dos y simplificar el cálculo de la dirección efectiva de las celdas.

La gran duda es si realmente acelera el cálculo de la dirección o no. Analiza cómo almacena las matrices C y cómo cambia el cálculo de la dirección efectiva de las celdas en el código en C al eliminar el padding (`PADDING = 0` en `sudoku_2021.h`). Evalúa para los distintos modos de optimización del compilador el cambio en rendimiento en uno y otro caso.

EVALUACIÓN DE LA PRÁCTICA

La práctica se presentará online **aproximadamente** el 14-15 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 22 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** disponible en la página web de la asignatura (moodle), 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. Este apartado os suele costar. El objetivo es remarcar aquellos mensajes que queráis que se queden en la cabeza del lector. Pensad que es lo último que va a leer. Algunos temas de los que podéis hablar son: ¿para qué ha servido vuestro trabajo?, ¿qué habéis conseguido?, ¿qué opináis de los resultados?, ¿y de la práctica que os hemos pedido?

Se valorará que el texto sea **claro y conciso**. Cuánto más fácil sea entender el funcionamiento del código y vuestro trabajo, mejor.

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, por ejemplo Memoria_p1_ NIP-Apellidos_Estudiante1_NIP-Apellidos_Estudiante2.pdf para el archivo de la memoria.