Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

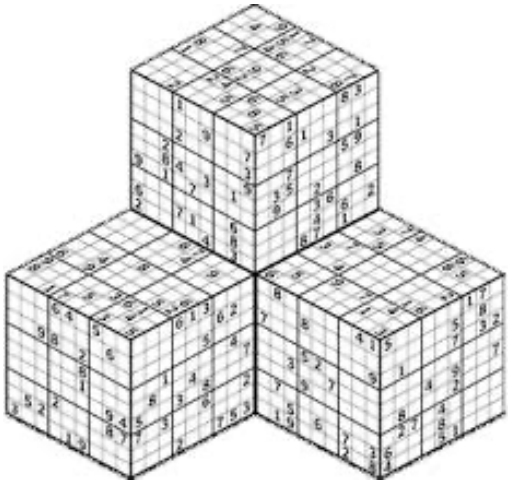
PROYECTO HARDWARE

3º DE GRADO EN INGENIERÍA INFORMÁTICA, CURSO 2021/2022

UNIVERSIDAD DE ZARAGOZA

PRÁCTICA 1

DESARROLLO DE CÓDIGO PARA EL PROCESADOR ARM



PRÁCTICA 1: DESARROLLO DE CÓDIGO PARA EL PROCESADOR ARM

Introducción 3

Objetivos 3

Conocimientos previos necesarios 4

Entorno de trabajo 4

Material disponible 4

Estructura de la práctica 4

Parte A: Ejemplo de desarrollo de código para el procesador ARM7 5

Parte B: Aceleración de un juego de Sudoku. 6

Trabajo a Realizar 7

¿Cómo funciona el código que nos han dado? 11

¿Cómo podemos medir tiempos e instrucciones? 12

Apartado opcional 1 14

Apartado opcional 2 14

Evaluación de la práctica 14

Anexo I. Algunos consejos de programación para optimizar el código 15

Anexo II. Realización de la memoria 17

Anexo III. Entrega de la memoria 17

INTRODUCCIÓN

Este documento es el guion para la realización de la primera práctica de la asignatura Proyecto Hardware del Grado en Ingeniería Informática de Escuela Universitaria Politécnica de Teruel de la Universidad de Zaragoza.

En esta primera práctica vamos a entender la funcionalidad del código suministrado, y optimizar el rendimiento de un juego acelerando las funciones computacionalmente más costosas (críticas). A partir del código facilitado, se desarrollará código en C y en ensamblador para un procesador ARM y se ejecutará sobre un emulador en el PC. Para ello, tendremos que aprender a trabajar con el entorno de desarrollo Eclipse + gcc.

Finalmente, se medirá el rendimiento respecto al código optimizado generado por el compilador y se documentarán los resultados. Más en detalle, se medirá el tamaño en bytes, el número de instrucciones ejecutadas, el tiempo de ejecución, y se verificará que todas las versiones del código den un resultado equivalente.

OBJETIVOS

- Interactuar con un microcrontrolador y ser capaces de **ejecutar** y **depurar**.
- Conocer la estructura segmentada en 3 etapas del procesador ARM7 que actualmente es uno de los más utilizados en los sistemas empotrados.
- Familiarizarse con el entorno Eclipse 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, adecuándolo 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.
- Ser capaces de depurar código en ensamblador, siguiendo el contenido de los registros internos del procesador y la memoria.
- Comprobar automáticamente que varias implementaciones de una función mantienen la misma funcionalidad.

CONOCIMIENTOS PREVIOS NECESARIOS

En esta asignatura el estudiante necesitará aplicar con soltura los contenidos impartidos en las asignaturas previas, en particular, Arquitectura y Organización de Computadores 1 y 2.

ENTORNO DE TRABAJO

Esta práctica se realiza con **Eclipse**, que está instalado en el laboratorio de la asignatura (y que también podéis instalaros en vuestros portátiles), junto con las herramientas de **gcc** para compilación cruzada y el complemento (*plug-in*) para simular procesadores ARM7TDMI.

MATERIAL DISPONIBLE

En el sitio web de la asignatura (en Moodle) puede encontrarse el siguiente material de apoyo:

- Cuadernos de prácticas de una asignatura de la Universidad Complutense que usa el mismo entorno de trabajo y las mismas placas.
    - ***GuiaEntorno.pdf***: Presenta el entorno de desarrollo que vamos a usar durante la asignatura. **Debéis repasarlo detenidamente.**
    - ***P1-EC.pdf***: Repasa algunos conceptos de C y del lenguaje ensamblador de ARM. **Debéis repasarlo detenidamente.**
- Breve resumen del repertorio de instrucciones ARM.
- Breve resumen del repertorio de instrucciones Thumb.
- Diapositivas resumen y Manual de la arquitectura ARM.
- Fuentes para los apartados A y B
- Directrices para redactar una memoria técnica, imprescindible para redactar la memoria de la práctica.

ESTRUCTURA DE LA PRÁCTICA

La práctica consta de dos partes, la primera es una toma de contacto con el entorno de trabajo y el procesador ARM7, mientras que en la segunda los alumnos deberán realizar un pequeño proyecto de desarrollo de código.

La práctica completa se debe presentar en un plazo determinado, y será la base para la práctica siguiente. Las fechas concretas se anunciarán debidamente en moodle.

PARTE A: EJEMPLO DE DESARROLLO DE CÓDIGO PARA EL PROCESADOR ARM7

DURACIÓN: 1 SEMANA

TRABAJO PREVIO. Antes de realizar esta práctica se debe repasar la arquitectura ARM y cómo trabajar en un entorno mixto C/ensamblador.

El objetivo de este apartado es desarrollar una función que copie veinticinco datos de una zona de memoria. Deben implementarse dos versiones distintas:

- C: se proporciona un esqueleto, al que deben añadirse algunas instrucciones
- Ensamblador ARM: se proporciona un esqueleto, al que deben añadirse algunas instrucciones.

La función tiene los siguientes parámetros:

- **src**: dirección de inicio de los datos a copiar
- **dst**: dirección de destino de los datos copiados

El estudiante deberá integrar las dos versiones de la función en el esqueleto que se proporciona y comprobar que el código se ejecuta correctamente (se recomienda no integrar una función hasta haber depurado la anterior). Se debe comparar el rendimiento de las dos implementaciones. Para ello, cada grupo deberá medir el tamaño del código de cada una de las dos versiones, así como el número de instrucciones que ejecuta cada una.

Los pasos a seguir para configurar el entorno de trabajo (*workspace*) están detallados en el archivo: "**GuiaEntorno.pdf**" (el código que se usa es distinto, pero los pasos a seguir son los mismos). En concreto, en la página 3 se indica la creación de un proyecto y en la página 16 "depuración sobre el simulador". Dadas las características concretas de nuestro laboratorio existen pequeñas diferencias:

- Es preciso añadir a la variable de entorno PATH el camino correcto al compilador (antes del punto 9 de la página 4) y al depurador (al final del punto 2 de la página 17). Este paso está detallado en la documentación publicada en la web de la asignatura.
- Si no usáis vuestro ordenador personal, no es recomendable trabajar en local (que es lo que indica la guía del entorno en el punto 2). Debéis trabajar en un directorio en el que tengáis permisos de lectura y escritura y trabajar con *paths* cortos (Eclipse no reconoce los *paths* largos). Por ejemplo D:\Workspace es una buena elección.

Este apartado es imprescindible realizarlo para poder continuar. Su objetivo es que la primera toma de contacto con ARM sea sencilla, y que el siguiente apartado resulte menos complejo. Además, tenéis que aprender a configurar los proyectos de Eclipse, dado que **el siguiente apartado se configurará de manera similar a éste**. Se recomienda que lo enseñéis al profesor para comprobar que lo habéis hecho todo bien.

PARTE B: ACELERACIÓN DE UN JUEGO DE SUDOKU.

DURACIÓN: 2 SEMANAS

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, el 8 y el 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.

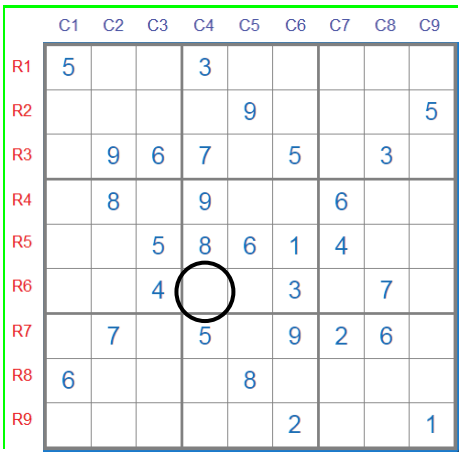


Figura 1. Ejemplo de resolución de Sudoku.

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 2a).

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 2b).

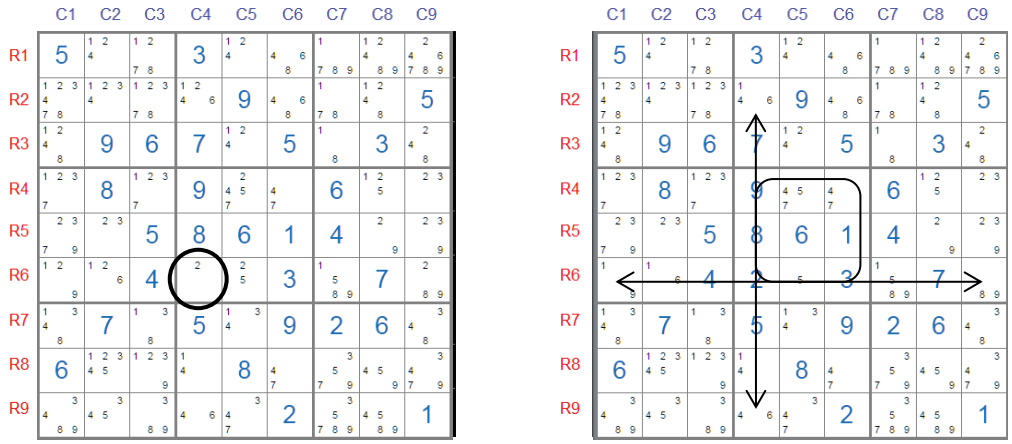


Figura 2. Actualización de candidatos tras inserción de valor.

TRABAJO A REALIZAR

Suponemos que cierta empresa quiere lanzar un sistema, llamado PH_sudoku, diseñado para facilitar que una persona resuelva el rompecabezas. PH_sudoku se ejecutará en un procesador ARM7 y permitirá al jugador 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.

A lo largo del presente curso, os encargaréis de la implementación de **PH_sudoku** realizando un prototipo sobre el sistema de desarrollo disponible. El dispositivo ayudará en las tareas de escaneo y marcado de las celdas, pero el análisis para la resolución va a recaer íntegramente sobre el jugador humano.

El usuario deberá ser capaz de introducir cualquier número del 1 al 9 en las celdas que no contengan una pista inicial, ya sea rellenando una celda vacía o modificando un valor previo. Deberá, así mismo, ser capaz de borrar un número de una casilla suya poniendo un cero (volverá al estado inicial). Para terminar la partida, el usuario introducirá un valor en la casilla no existente R0C0. Tras cada movimiento, el sistema deberá comprobar que no hay errores y recalcular la lista de candidatos de cada celda. Básicamente:

- Si la celda está vacía, comprobará el valor con la lista de candidatos. Si no hay errores, propagará en la fila, la columna y su región el nuevo valor tachándolo de las listas de candidatos de las distintas celdas.
- Si la celda no está vacía (y no es una pista) y el valor es válido, será necesario volver a recalcular las listas de candidatos de todas las celdas de la cuadrícula.

Como primer paso, han escrito una versión beta del programa en C (como versión beta, está sujeta a fallos. Avisad si veis alguno). El dispositivo PH_sudoku debe tener un coste reducido, por ello están preocupados por las necesidades computacionales y tiempo de ejecución, y os piden que lo reduzcáis. En esta primera práctica nos centraremos en la implementación eficiente de la función más crítica del código: **`candidatos_actualizar_c()`**.

También se plantean incluir una pequeñísima memoria muy rápida en la que guardar estas funciones, por lo que es importante conseguir que ocupen lo mínimo posible.

Por tanto, tenéis que:

a) **PASO 1: 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 la definición de las funciones y el pseudocódigo del comportamiento de ambas funciones. Debéis **implementar ambas en C, verificando** sobre la cuadrícula en memoria **la correcta ejecución**, comprobando a mano que el cálculo de los candidatos de toda la cuadrícula ha sido realizado satisfactoriamente.

b) **PASO 2: Revisar la información facilitada por el compilador**, si hay errores o avisos, entended qué ocurre y corregidlo. En particular, deberéis comprobar los tipos utilizados para definir las distintas variables.

c) **PASO 3: 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.

d) **PASO 4: Implementar la función `candidatos_propagar_arm()` en ensamblador ARM** para tratar de mejorar el rendimiento y el tamaño del código. Realiza un programa que llame a esta nueva función desde C.

   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.

e) **PASO 5: Realizar `candidatos_actualizar_arm()` en ensamblador ARM.**

f) **PASO 6: Modificar el código para poder ejecutar las cuatro posibles combinaciones (C-C, C-ARM, ARM-C y ARM-ARM)**.

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()`; no se puede eliminar la llamada a ninguna función, o eliminar el paso de parámetros; y no 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, ésta debe ser una llamada convencional a una función. Es decir, hay que pasar los parámetros, utilizando el estándar **ATPCS** (*ARM-Thumb Procedure Call Standard*), a través de los registros correspondientes. No sirve hacer un salto sin pasar parámetros.

Para las llamadas a ARM se debe crear un marco de pila (o bloque de activación), tal y como se explica en las transparencias de la asignatura. 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.

g) **PASO 7: Realizar una nueva función candidatos_actualizar_all(),** que fusione las 2 funciones en ensamblador realizadas anteriormente.

   Tened en cuenta, que 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 el código (inlining) de la misma en vuestra función en ARM. De esa forma nos evitamos llamar a la subrutina y las sobrecargas asociadas a esta operación (creación y destrucción del Bloque de Activación, preservación de registros, etc.).

h) **PASO 8: 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.

   En total tenemos 5 configuraciones distintas (2 para actualizar * 2 de propagar + actualizar_all). Vuestro código debe recalcular la cuadrícula inicial para cada una de las 5 combinaciones y **verificar de forma automática que todas las configuraciones 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.

i) **PASO 9: Medidas de rendimiento**. Queremos evaluar el tiempo de ejecución de cada una de estas configuraciones, pero aún no tenemos el sistema sobre el que finalmente se ejecutará y no disponemos de una manera fina de medir tiempo. La ejecución se está realizando sobre un simulador en un ordenador con instrucciones Intel x86, mientras realiza otras múltiples tareas. Pero podemos hacernos una idea si medimos manualmente el tiempo de ejecución. **Calcular el tiempo de ejecución de las 5 combinaciones y razonar los resultados obtenidos.** Más adelante en el guion os mostramos una forma de medir los tiempos.

   Por defecto, el compilador de C genera un código seguro y fácilmente depurable (*debug*). Esta primera versión es buena para ejecutar paso a paso, 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[1]. Estas configuraciones se suelen resumir bajo los flags `–O0, -O1, -O2, -O3, -Os`.

   Estudiad el impacto en el rendimiento del código C de las optimizaciones (`-O1, -O2`…) y comparadlo con las otras versiones en rendimiento, tamaño y número de instrucciones ejecutadas.

   A la hora de evaluar este trabajo se valorará especialmente que cada grupo haya sido capaz de optimizar el código. **Cuanto más 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.

[1] Os sorprenderíais al saber la cantidad de software comercial generado sin optimizar y en modo *debug*.

¿CÓMO FUNCIONA EL CÓDIGO QUE NOS HAN DADO?

La información del tablero está guardada en una estructura de datos denominada `cuadricula`, definida en `tableros.h`, consistente en una matriz de celdas. 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.

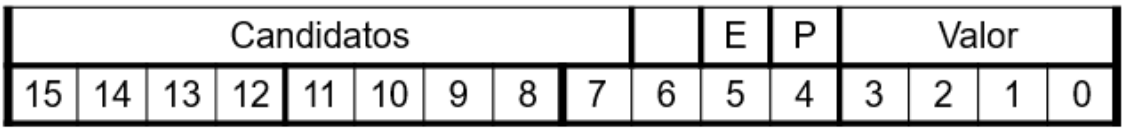


Figura 3. Codificación de campos en 16 bits.

EJEMPLO:
0x9c13 = 100111000 0 0 1 0011

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, 0011 indica que la celda contiene un 3.

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 cuadrícula sea potencia de 2. Esto hace que las cuadrículas ocupen más espacio en memoria, pero simplifica la gestión de direcciones y podría acelerar el código.

Buscad en vuestro código el *array* `cuadricula` (por ejemplo, podéis ir ejecutando paso a paso y mirando los valores de los registros, o poner un **`breakpoint`** en la línea de código de interés) y poned un monitor de memoria en esa dirección (menú *view / memory Windows*). Representad esos datos en memoria en formato ***`Hex Integer`*** (pulsad el botón derecho sobre el monitor y elegid ***`format`***, ver Figura 4). Para simplificar la visualización en el entorno, `cuadricula` se ha definido de 9 filas por 16 columnas, las 9 primeras contienen valores válidos de celdas, el resto se han añadido para alinear el tablero. Configuradlo tal y como aparece en la Figura 5. Se deben ver 16 columnas. Si la dirección asignada por el enlazador no es múltiplo de 32 el tablero 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.

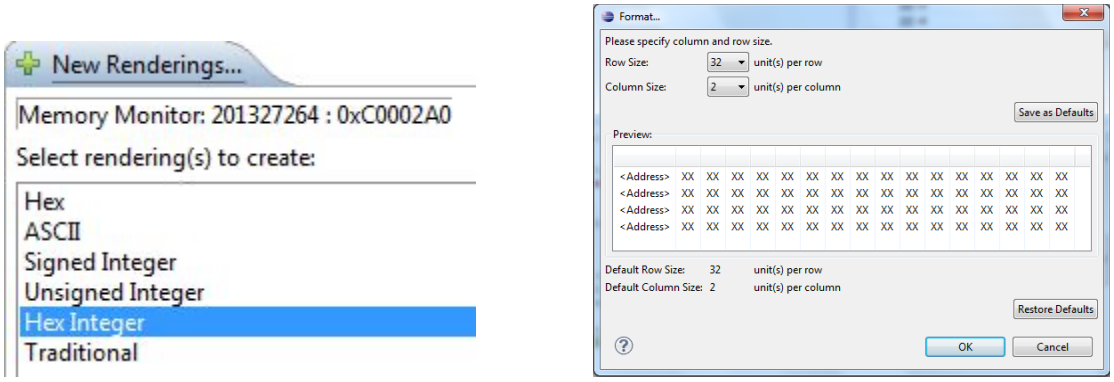


Figura 4. Configurando el tablero en Eclipse

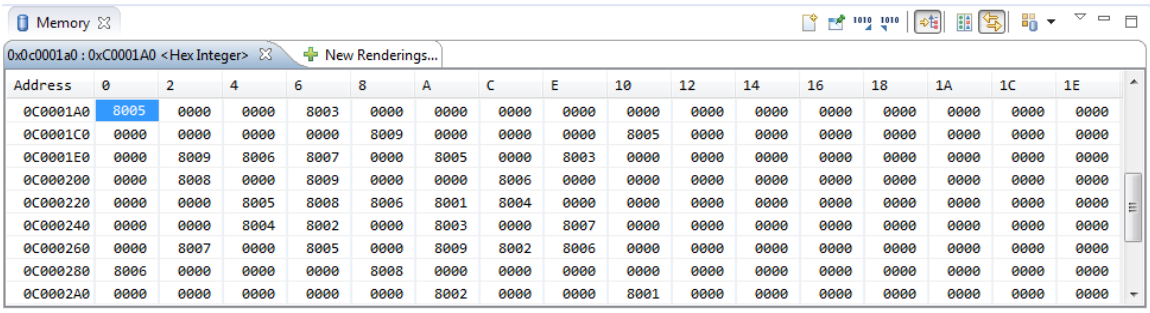


Figura 5. Cuadrícula inicial tal y como debe verse

¿CÓMO PODEMOS MEDIR TIEMPOS E INSTRUCCIONES?

Medición del número de instrucciones

Contar el número de instrucciones puede resultar algo tedioso. Utilizando comandos del depurador podemos simplificarlo. En la pestaña de la **consola de depuración** (normalmente abajo a la izquierda) nos muestra información sobre la ejecución del depurador y también nos permite introducir comandos. Por ejemplo, si tecleamos `stepi` (*step in*) se realizará un paso de ejecución.

Una forma de contar las instrucciones es parar la ejecución antes de la primera instrucción a contar y anotar el PC de la última que queremos contar. A continuación, crearemos e inicializaremos una variable *contador* con el siguiente comando:

```
set $count=0
```

Ya estamos preparados para ir ejecutando el código hasta el PC de la última instrucción mientras se cuentan las instrucciones. Realizaremos un pequeño bucle mediante el siguiente comando:

```
while ($pc != 0xpc_ultima)
```

Nos saldrá como un pequeño cursor sobre el que iremos introduciendo las órdenes que queremos dentro del bucle, en nuestro caso, ejecutar paso a paso e ir incrementando el contador:

```
stepi
set $count=1+$count
end
```

Según el número de instrucciones, el simulador tardará más o menos en procesar el bucle. De hecho, puede tardar varios segundos. Cuando acabe, solo nos quedará ver cuánto vale el contador.

```
print $count
```

Medición del tiempo de ejecución

La ejecución completa del cálculo del tablero en cualquiera de las cinco combinaciones es relativamente rápida. Si queremos **medir manualmente el tiempo** de la ejecución no nos queda otro remedio que ejecutarla múltiples veces seguidas. Por ejemplo, podemos ponerla en un bucle para que se ejecute mil veces y dividir por mil el tiempo de ejecución del bucle completo, para así obtener el tiempo de una ejecución. Aun así, como trabajamos sobre un sistema donde se están ejecutando simultáneamente múltiples tareas que no controlamos, conviene realizar esta medida varias veces para verificar que es consistente.

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. Mirad el manual y el código en ensamblador generado por el compilador. Explicad qué técnicas ha aplicado en cada versión y cómo las ha empleado.

Buscad optimizar vuestras funciones ARM aplicando las mismas técnicas. ¿Sois capaces 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. ¿Sois capaces de optimizar las funciones en C para ayudar al compilador a generar "mejor" código?

APARTADO OPCIONAL 2

Como ya se ha comentado, las cuadrículas 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 **aproximadamente** el 21 de octubre.

La memoria habrá que presentarla **aproximadamente** el 26 de octubre.

Las fechas definitivas se publicarán en la página web de la asignatura (*moodle*).

ANEXO I. ALGUNOS CONSEJOS DE PROGRAMACIÓN PARA OPTIMIZAR EL CÓDIGO

A la hora de evaluar este trabajo se valorará especialmente que cada grupo 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 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 operaciones 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 sean aplicables, pero muchas sí que lo serán y os animamos a utilizarlas.

ANEXO II. REALIZACIÓN DE LA MEMORIA

La memoria de la práctica tiene diseño libre, pero es obligatorio que incluya los siguientes puntos:

1. Resumen ejecutivo (una cara como máximo). El resumen ejecutivo puede ser considerado como 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. Marcos de pila.
3. Código fuente comentado del apartado B. Además, cada función debe incluir una cabecera en la que se explique cómo funciona, qué parámetros recibe, así como dónde los recibe y para qué usa cada registro (por ejemplo, en el registro 4 se guarda el puntero a la primera matriz…).
4. Descripción de las optimizaciones realizadas al código ensamblador.
5. Resultados de la comparación entre las distintas versiones de las funciones.
6. Descripción de los problemas encontrados en la realización de la práctica y sus soluciones.
7. Conclusiones

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

ANEXO III. ENTREGA DE LA MEMORIA

La entrega de la memoria será a través de la página web de la asignatura (*moodle* en https://moodle.unizar.es/add/course/view.php?id=46338). 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