29-10-2021

David Ros y Álvaro Fraidias

Práctica 1

Desarrollo de código para el procesador ARM

Contenido

[1. Resumen ejecutivo 3](#_Toc86427144)

[2. Introducción 3](#_Toc86427145)

[3. Objetivos 3](#_Toc86427146)

[4. Apartado A 3](#_Toc86427147)

[4.1 Inicio y variables 3](#_Toc86427148)

[4.2 Código en C 4](#_Toc86427149)

[4.2.1 Tiempo ejecución 4](#_Toc86427150)

[4.3 Código en ensamblador ARM 5](#_Toc86427151)

[4.3.1 Tiempo ejecución 6](#_Toc86427152)

[4.4 Comparación C y ARM 6](#_Toc86427153)

[5. Parte B 7](#_Toc86427154)

[5.1 Introducción 7](#_Toc86427155)

[5.2 Bloque de activación 7](#_Toc86427156)

[5.3 Pila 8](#_Toc86427157)

[5.3 AAPCS 8](#_Toc86427158)

[5.3.1 Reglas de uso 8](#_Toc86427159)

[5.3.2 Reglas uso de la pila 9](#_Toc86427160)

[5.4 Parte C 9](#_Toc86427161)

[5.4.1 Cabecera e inicio 9](#_Toc86427162)

[5.4.2 Candidatos\_actualizar\_C 10](#_Toc86427163)

[5.4.3 Candidatos\_propagar\_c 11](#_Toc86427164)

[5.4.3.1 Declaración variables e inicialización 11](#_Toc86427165)

[5.4.3.2 Recorrer fila 11](#_Toc86427166)

[5.4.3.3 Recorrer columna 11](#_Toc86427167)

[5.4.3.4 Recorrer región 12](#_Toc86427168)

[5.5 Parte ARM 13](#_Toc86427169)

[5.5.1 Cabecera e inicio 13](#_Toc86427170)

[5.5.2 Candidatos\_actualizar\_arm 14](#_Toc86427171)

[5.5.3 Candidatos\_propagar\_arm 16](#_Toc86427172)

[5.5.3.1 Declaración de variables e inicialización 16](#_Toc86427173)

[5.5.3.2 Recorrer fila 17](#_Toc86427174)

[5.5.3.3 Recorrer columna 17](#_Toc86427175)

[5.5.3.4 Recorrer región 18](#_Toc86427176)

[5.5.4 Candidatos\_propagar\_all 19](#_Toc86427177)

[5.6 Medidas de rendimiento 21](#_Toc86427178)

[6. Automatización de la solución 21](#_Toc86427179)

[7. Errores 22](#_Toc86427180)

[8. Conclusión 24](#_Toc86427181)

[9. Bibliografía 24](#_Toc86427182)



# 1. Resumen ejecutivo

En el siguiente pdf se define el famoso juego sudoku para más tarde realizar la función más crítica de dicho juego: actualizar el tablero cuando se coloca un número. Una vez definido el problema se realiza la solución implementada tanto en C como en ARM para luego ir intercambiando código de una arquitectura a otra midiendo el rendimiento.

# 2. Introducción

Esta práctica se divide en dos apartados cuyos esqueletos son proporcionados por el profesor y debemos hacer un previo estudio para entenderlos: El primero consiste en implementar tanto en C como en ARM un programa que copie 25 valores de una dirección de memoria a otra y el segundo consiste en optimizar el rendimiento del juego “Sudoku” acelerando las funciones computacionalmente más costosas (críticas). Más adelante explicaremos con más detalle cada apartado.

# 3. Objetivos

Los objetivos de esta práctica son:

* Instalación y configuración correcta del entorno de desarrollo “Eclipse”.
* Volver a recordar el lenguaje de programación C y ARM.
* Entender la finalidad y el funcionamiento del estándar ATPCS.
* Aprender a desarrollar y optimizar código en ensamblador ARM y C.
* Comprobar la interoperabilidad entre C y ARM.

# 4. Apartado A

En este apartado se desarrollará un código tanto en C como en ARM para trasladar un bloque de datos de una dirección de memoria a otra.

## 4.1 Inicio y variables

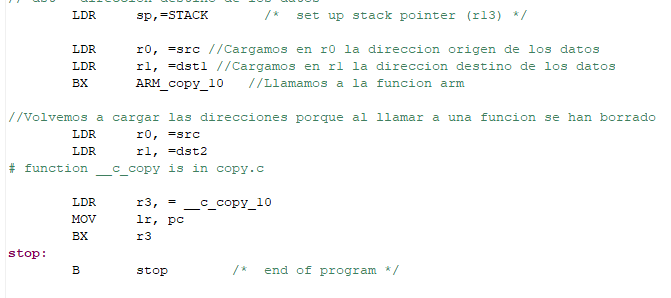


Ilustración 1:Inicio del programa

Como se puede ver en la ilustración 1, cargamos las direcciones de memora origen y destino para llamar a la función en ARM. Después volvemos a cargar la dirección origen y la otra dirección destino y llamamos a la función en C.

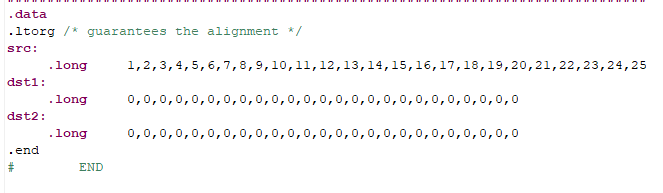


Ilustración 2: Datos utilizados en el programa

## 4.2 Código en C

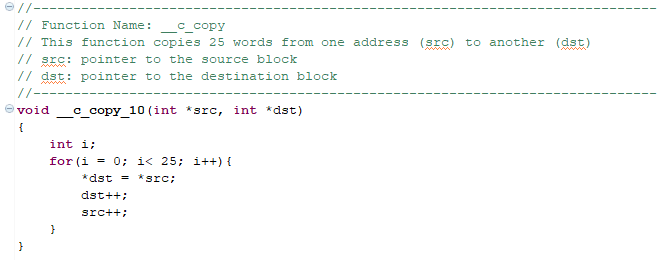


Ilustración 3. Código parte A en c

En la ilustración 3 se representa el método necesario para mover un bloque de datos de una zona de memoria a otra. Dicho método recibe dos punteros como parámetros: “src” apunta al bloque donde inicialmente están los datos que se pretenden mover y “dst” apunta al bloque donde finalmente se van a guardar los datos. Mediante un bucle iterado 25 veces se iguala el puntero del bloque destino al puntero del bloque inicial. Acto seguido se incrementa cada puntero para acceder a la siguiente posición de memoria.

### 4.2.1 Tiempo ejecución

Para calcular el tiempo de ejecución hemos hecho un bucle que se ejecuta 1000 veces y hemos medido el tiempo manualmente.

|  |  |  |
| --- | --- | --- |
| Programa | Número de instrucciones | Tiempo de ejecución(milisegundos) |
| \_\_c\_copy\_10 | 76 | 9,8 |

## Código en ensamblador ARM

En la ilustración 4 se representa el código en ensamblador cuya funcionalidad es la misma que el de la ilustración 1. En este caso utilizamos un bucle for donde se iterará 4 veces en las cuales en cada iteración se copiarán 9 palabras. Gracias a LDM y STM podemos cargar o almacenar varias palabras simultáneamente.

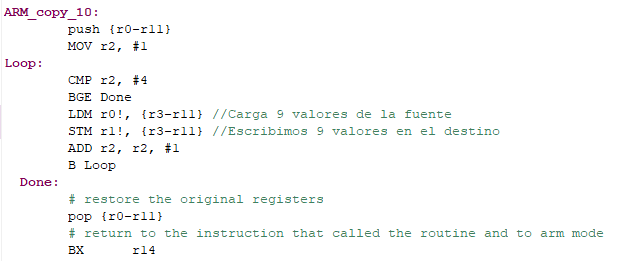


Ilustración 4. Código parte A en ensamblador ARM

Como las direcciones destino de ambos métodos son contiguas, en la última iteración de este código sobrescribe 2 valores en la dirección destino de C. Esto supondría un error si el método en C leyese de memoria ya que estaría leyendo un valor erróneo. Como escribe no pasa nada ya que sobrescribe el valor.

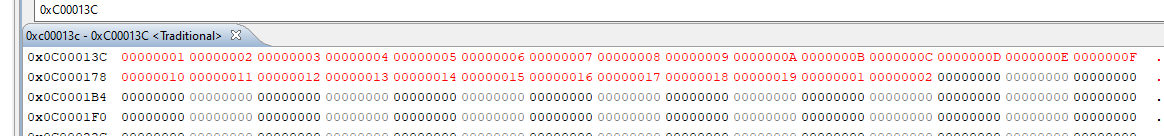


Ilustración 5. Demostración de que en la última iteración copia 2 valores más de la cuenta

### 4.3.1 Tiempo ejecución

Igual que en el caso anterior, para calcular el tiempo de ejecución hemos hecho un bucle que se ejecuta 1000 veces y hemos medido el tiempo manualmente.

Al hacerlo nos hemos dado cuenta de que necesitamos dos registros:

* Uno para iterar el bucle
* Otro para comparar y saber cuándo se ha llegado al final del bucle.

Por consiguiente, debemos copiar menos palabras en cada iteración (exactamente 5 valores).

El código generado es el siguiente:

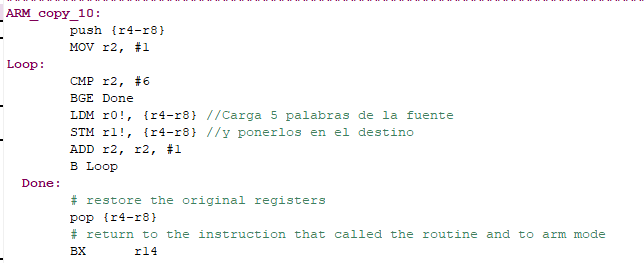


Ilustración 6. Bucle ARM parte A

## Comparación C y ARM

Se puede observar que el número de instrucciones y por lo tanto el tiempo de ejecución del programa en ARM tiene valores inferiores ya que en ARM utilizamos las instrucciones de carga y almacenamiento múltiple de forma que podemos cargar y almacenar 5 datos a la vez lo cual nos permite ejecutar el bucle sólo 5 veces. Sin embargo, el programa en C tiene un bucle que se ejecuta 25 veces y por lo tanto es menos eficiente.

# 5. Parte B

## 5.1 Introducción

En esta parte se pretende desarrollar la función crítica del juego Sudoku llamada “candidatos\_actualizar” que como bien dice su nombre se encarga de recorrer el tablero comprobando si en cada celda hay un valor y en caso de haberlo, propagar ese valor para que las demás celdas lo eliminen de sus valores candidatos.

En caso de ser necesario propagar, se llamará desde “candidatos\_actualizar” a otra función llamada “candidatos\_progapar” que propagará ese valor por toda la fila, columna y región de la celda protagonista.

A la hora de realizar la práctica se ha diseñado estos dos métodos tanto en C como en ARM para acto seguido, combinar todas las posibilidades: C\_C, C\_ARM, ARM\_C y ARM\_ARM y evaluar sus rendimientos. También se ha hecho una función en ARM llamada “candidatos\_arm\_all” que combina dentro de ella la función “candidatos\_actualizar\_arm” y “candidatos\_propagar\_arm” y una función llamada “cuadricula\_candidatos\_verificar” para verificar que el tablero resultante es correcto.

## 5.2 Bloque de activación

Salvo que la subrutina pueda realizar las tareas utilizando exclusivamente los registros destinados al paso de parámetros, será necesario crear y gestionar un espacio de memoria donde la subrutina pueda almacenar la información que necesita durante su ejecución. A este espacio de memoria se le denomina bloque de activación de la subrutina. El bloque de activación de una subrutina cumple los siguientes cometidos:

* En el caso que la subrutina llame a otras subrutinas, almacenar la dirección de retorno original.
* Proporcionar espacio para las variables locales de la subrutina.
* Almacenar los registros que la subrutina necesita modificar y que el programa que ha hecho la llamada no espera que sean modificados.
* Mantener los valores que se han pasado como argumentos a la subrutina.

## 5.3 Pila

Para implementar el bloque de activación necesitamos una estructura de datos conocida como pila. Para implementar dicha pila debemos reservar espacio en la memoria e indicarle al registro 13 (registro sp) la dirección de inicio de dicho espacio. En nuestro caso lo hacemos al inicio del programa ejecutando la siguiente instrucción:



Ilustración 7. Carga de la dirección pila en el r13

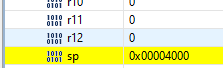


Ilustración 8. Ejecución de la ilustración 7

## 5.3 AAPCS

AAPCS son las siglas de “ARM Architecture Procedure Call Standard”. Este estándar implementa una serie de reglas que explicaremos en los siguientes subapartados.

### 5.3.1 Reglas de uso

* Los parámetros se pasan entre subrutinas a través de los registros R0-R3. La subrutina llamada no necesita restaurar el contenido de los registros R0 a R3 antes de regresar.

En nuestro caso hemos almacenado los registros R0-R3 en la pila ya que dentro de las subrutinas llamamos a otras subrutinas por lo que necesitamos recuperar los valores. También hemos usado algunos de estos registros como registros auxiliares.

* En la subrutina, se usan los registros R4-R11 para guardar las variables locales. El valor de estos registros debe guardarse al inicio de la subrutina y restaurarse al final.
* El registro R12 se utiliza como registro temporal intermedio para la llamada de procedimiento y se indica como IP.
* El registro R13 se utiliza como puntero de pila y se denota como SP. El valor del registro SP al entrar en la subrutina y el valor al salir de la subrutina deben ser iguales.
* El registro R14 se denomina registro de conexión y se denota como LR. Se utiliza para guardar la dirección de retorno de la subrutina. Si la dirección de retorno se guarda en la subrutina, el registro R14 se puede utilizar para otros fines.
* El registro R15 es el contador del programa, denominado PC. No se puede utilizar para otros fines.

### 5.3.2 Reglas uso de la pila

ATPCS estipula que la pila es de tipo FD (Descendente completo donde sp apunta al último valor introducido y la pila de datos crece de la dirección alta a la dirección baja), es decir, la pila de disminución completa y la operación de la pila es una alineación de 8 bytes. Por tanto, los comandos de uso frecuente son STMFD y LDMFD.

En nuestro caso nos dimos cuenta que daba igual usar las instrucciones STDMD y LDMFD o las instrucciones PUSH y POP. Sin embargo, hemos usado las instrucciones STDMD y LDMFD para ceñirnos exactamente al estándar.

## 5.4 Parte C

### 5.4.1 Cabecera e inicio

En esta parte de implementación en C se ha tenido que añadir una función “main” la cual es llamada desde el inicio del programa ARM. En esta función “main” se llama a la función “sudoku9x9”.

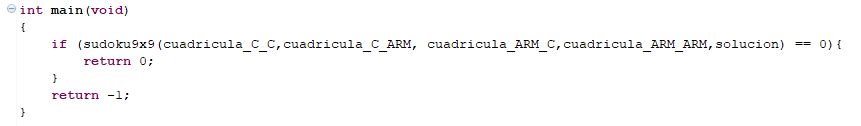


Ilustración 9. Función “main”

La función “sudoku9x9” tiene como parámetros todas las cuadrículas de las diferentes configuraciones, así como la cuadrícula solución. En esta función se declara un vector de celdas vacías con 5 posiciones, y se llama a cada una de las configuraciones para guardar el número de celdas vacías en cada una de las posiciones del vector.

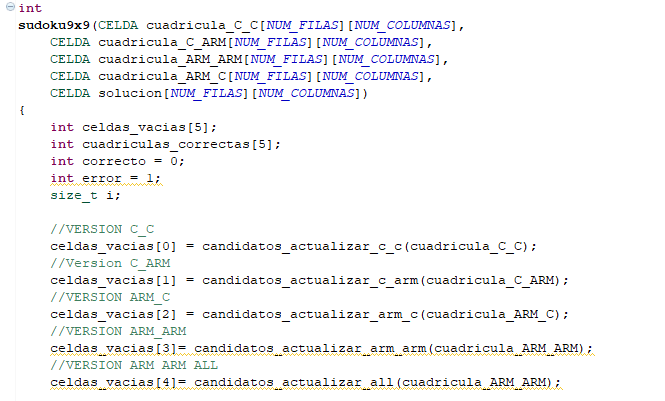


Ilustración 10. Función “sudoku9x9”

### 5.4.2 Candidatos\_actualizar\_C

De esta función existen dos versiones, una que llama a la función propagar implementada en C y otra que llama a la función propagar implementada en ARM. Sus implementaciones son idénticas salvo una llamada a una funcion función (candidatos\_propagar) y por lo tanto sólo se va a explicar la implementación de “candidatos\_actualizar\_c\_c”.

Dicha función, mostrada en la ilustración 11, realiza 2 bucles anidados, que recorren las filas y las columnas de la cuadrícula que se ha pasado a la función como parámetro. Una vez que se están recorriendo las filas y las columnas se almacena el valor de la celda correspondiente llamando a la función “celda\_leer\_valor”. Si el valor obtenido de esta función es 0 significa que no hay ningún valor en esa celda, y, por lo tanto, debemos actualizar el contador de casillas vacías. Si el valor obtenido es distinto de 0 quiere decir que existe un valor en dicha celda y debe ser propagado haciendo uso de la función “candidatos\_propagar\_c”.

Por último, esta función devuelve el número total de celdas vacías de la cuadrícula.

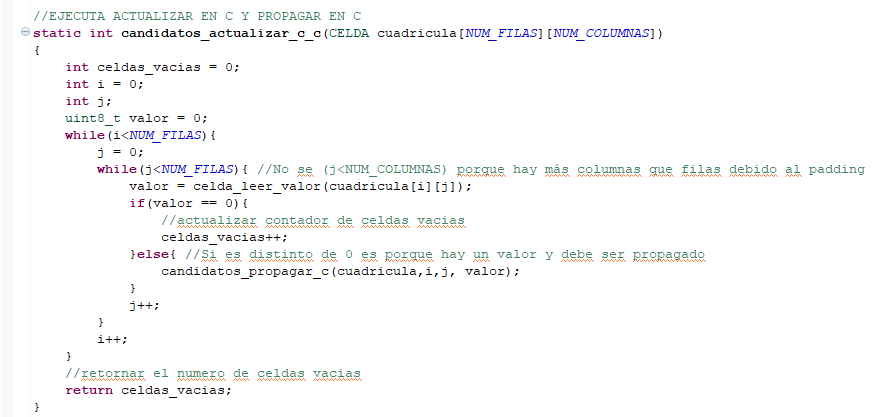


Ilustración 11. Función “candidatos\_actualizar\_c\_c”

### 5.4.3 Candidatos\_propagar\_c

Esta función se encarga de propagar el valor dado como parámetro por la fila, columna y región del tablero pasado también por parámetro.

Para explicar el funcionamiento debemos dividir el código en 3 partes:

#### 5.4.3.1 Declaración variables e inicialización

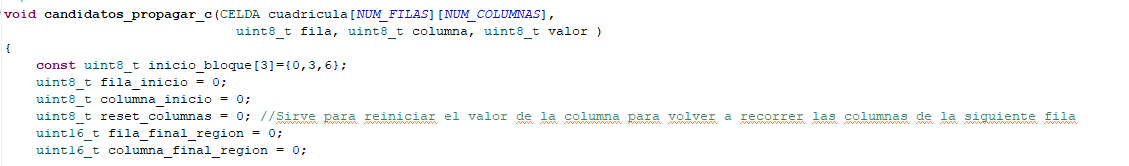


Ilustración 12: Declaración variables "candidatos\_propagar\_c"

La única variable que puede resultar rara de entender a simple vista es “inicio\_bloque” ya que es una estructura que nos sirve para establecer la fila y columna inicial de la región donde se encuentra el valor que queremos recorrer.

#### 5.4.3.2 Recorrer fila

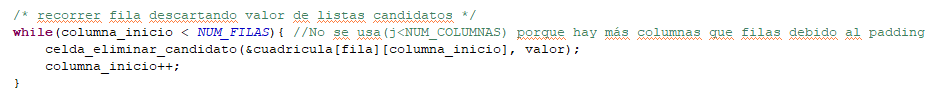


Ilustración 13: Bucle para recorrer las filas del tablero

Mediante un bucle recorremos las filas del tablero eliminando los candidatos.

#### 5.4.3.3 Recorrer columna

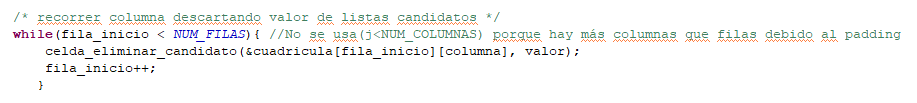


Ilustración 14: Bucle para recorrer las columnas del tablero.

Mediante un bucle recorremos las columnas del tablero eliminando los candidatos. En este caso no usamos la variable NUM\_COLUMNAS ya que hay más columnas que filas debido al padding. El padding consiste en poner 0 en las filas de la cuadricula a la hora de guardarlo en la memoria para poder ver en memoria el tablero correctamente.

#### 5.4.3.4 Recorrer región

Para recorrer la región primero debemos saber en qué región nos encontramos. Para entender cómo saber en qué región nos encontramos para recorrerla lo hemos explicado mediante el siguiente dibujo:

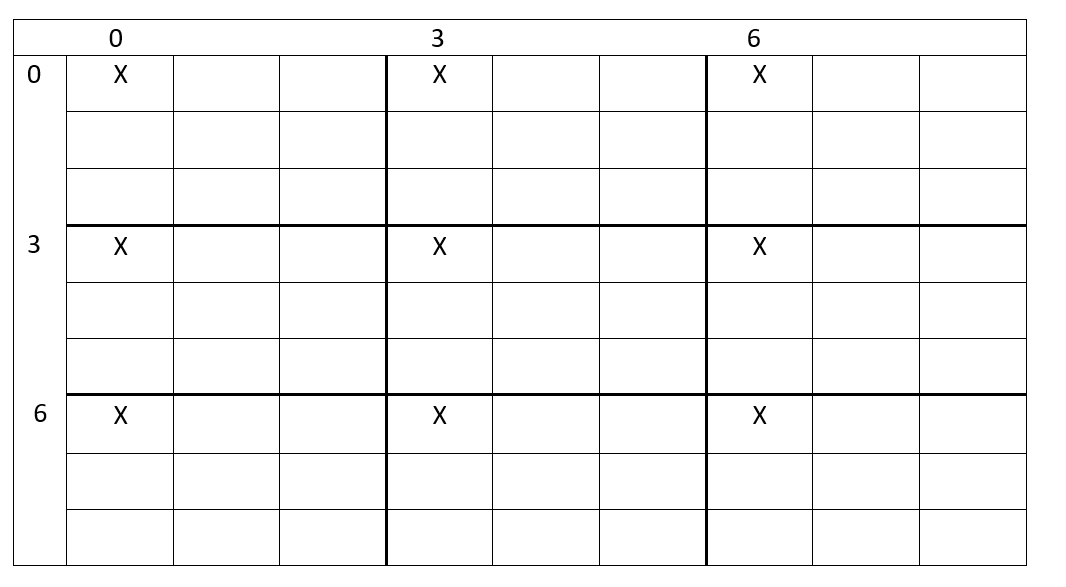


Ilustración 15: Dibujo para saber en qué región nos encontramos

Para recorrer la región debemos calcular la primera posición de la región en la que estamos, para ello usamos la fila y la columna que nos pasan como parámetro y guardamos en la pila.

Comparamos la fila y la columna con la fila y columna de la primera celda de cada región para saber si es mayor, menor o igual. De esta manera obtenemos la primera celda de la región que tenemos que recorrer.

Esta solución es menos eficiente que alguna de las propuestas en clase porque volvemos a recorrer 6 celdas que ya han sido revisadas.

Una vez explicado como sabemos que región recorrer vamos al código:

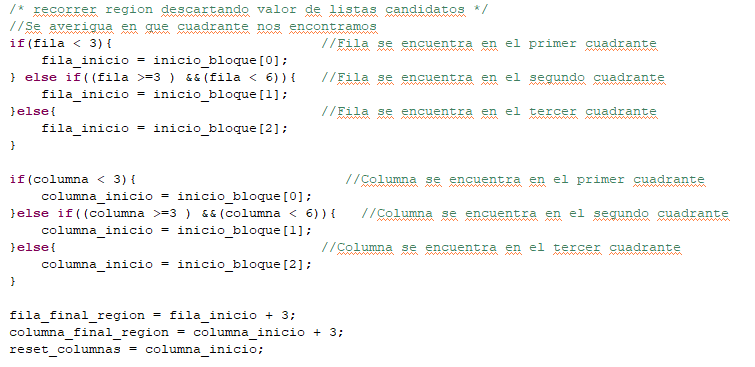


Ilustración 16: Averiguamos en que región estamos

En la ilustración 16 calculamos el inicio de la región a calcular y el final de dicha región. Después de esto, recorremos un doble bucle como se puede apreciar en la ilustración 17.

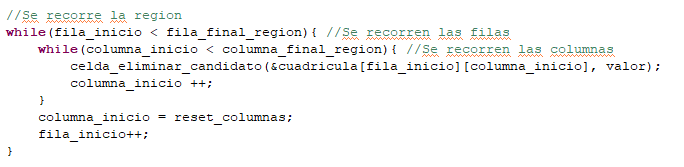


Ilustración 17. Función “candidatos\_propagar\_c”

## 5.5 Parte ARM

En esta parte simplemente se traduce el código de C del subapartado anterior.

### 5.5.1 Cabecera e inicio

Para poder utilizar las funciones definidas en ARM desde C hay que escribir en la cabecera del fichero de ARM la palabra reservada “global” junto con el nombre de la subrutina que deseamos llamar desde C. Gracias a esto, estas subrutinas pueden ser llamadas desde otro fichero.

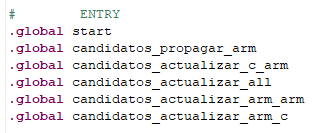


Ilustración 18. Declaración de las subrutinas para poder usarlas desde otro fichero

En cuanto al programa principal es simple: se carga la función de main en un registro, se guarda la dirección actual en el registro lr para retornar cuando haya acabado el programa y salta al main (función implementada en C).

Si nos damos cuenta el salto se realiza con la etiqueta “X” lo que significa que realizará en modo Thumb (16 bits).

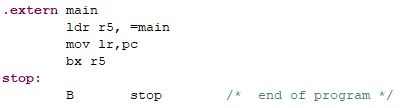


Ilustración 19. Inicio programa

### 5.5.2 Candidatos\_actualizar\_arm

Hay dos versiones de esta función:

Una que llama a la función propagar implementada en C y otra que llama a la función propagar implementada en ARM.

Las dos son iguales salvo la llamada a esta función, es decir, son iguales salvo una instrucción por lo que no tiene sentido explicar las dos por separado ya que hacen lo mismo. Por ello cogemos de ejemplo la subrutina “candidatos\_actualizar\_arm\_c” para explicarla:

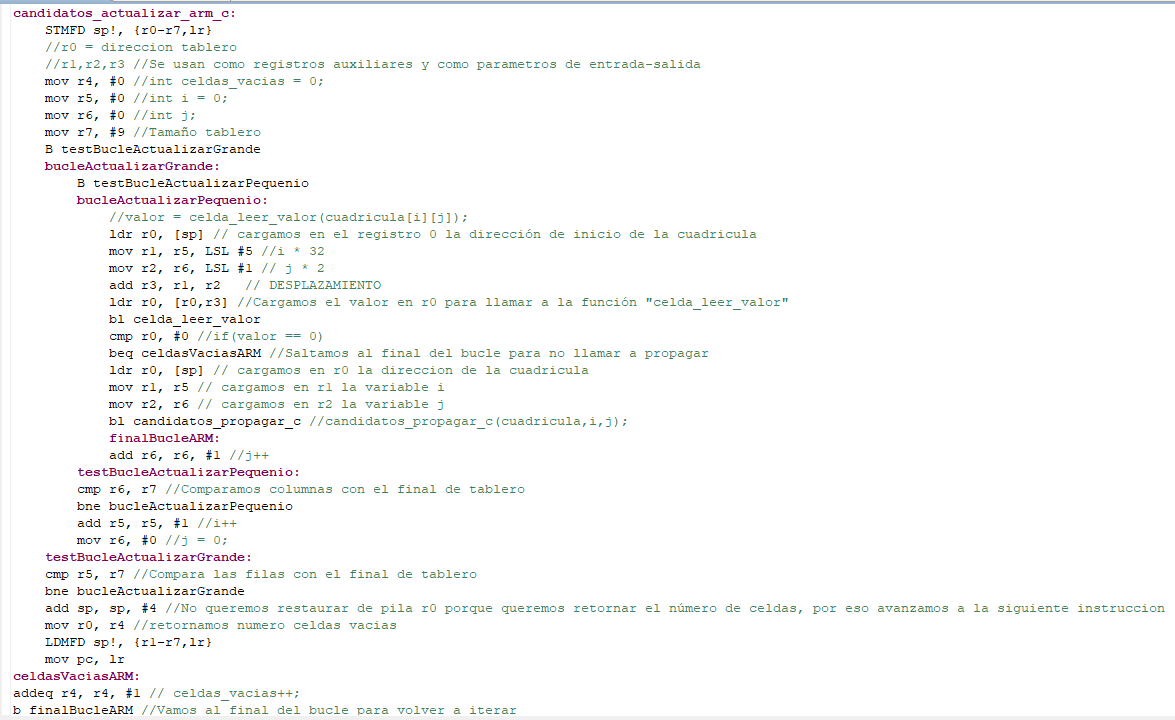


Ilustración 20. Subrutina “candidatos\_actualizar\_arm\_c” implementada en ARM.

Para explicarla bien se va a comentar paso por paso:

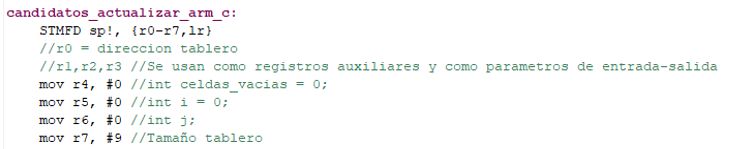


Ilustración 21. Variables "candidatos\_actualizar\_arm\_c"

Son necesarios un mínimo 7 registros (es la solución más optima ya que lo hemos revisado detenidamente una serie de veces y no hemos podido hacer esta subrutina con menos registros).

Los registros apilados en la pila quedarían de la siguiente manera:

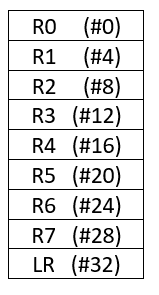


Ilustración 22: Pila con el bloque de activación de la subrutina "candidatos\_actualizar\_C" (sp apunta a r0)

Una vez explicada la creación de variables e inicialización pasamos a explicar el cuerpo de la subrutina:

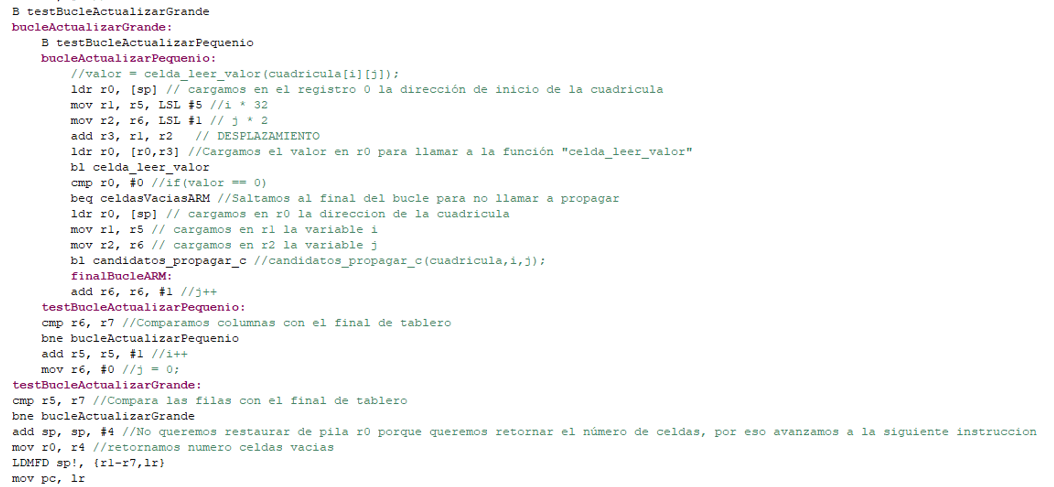


Ilustración 23. Bucle “candidatos\_actualizar\_arm\_c”

Para recorrer todo el tablero se necesita hacer un doble bucle (uno para las filas y dentro de él, uno para las columnas). Las condiciones de los dos bucles deben ser iguales ya que el tablero es un cuadrado que tiene las mismas filas que columnas.

La ejecución dentro del bucle es simple: se carga en el registro r0 la dirección de memoria de la celda que se quiere saber el valor (se carga en r0 para usarlo como parámetro al llamar a la función “celda\_leer\_valor”) y se comprueba si hay valor. En caso de haber valor llama a la función propagar y en caso de no haber valor a la función “celdaVacía” para que sume en celdas vacías e itere.

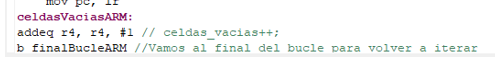


Ilustración 24. Subrutina “celdasVacíasARM”

Como se ha dicho anteriormente, esta función simplemente aumenta en 1 la cantidad de celdas vacías del tablero y pone en pc la dirección de retorno del final del bucle para que itere.

### 5.5.3 Candidatos\_propagar\_arm

Esta subrutina junto con “candidatos\_propagar\_all” son las subrutinas más largas del proyecto por lo que la dividiremos en 4 partes para comentarlas mejor: declaración de las variables, recorrer fila, recorrer columna y recorrer región.

#### 5.5.3.1 Declaración de variables e inicialización

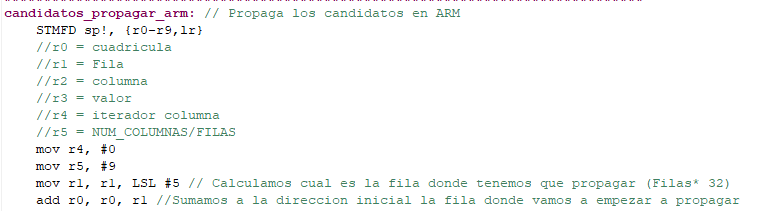


Ilustración 25. Declaración de las variables que vamos a usar

Los registros r8 y r9 se explicarán más adelante cuando los usemos.

La pila quedaría de la siguiente manera:

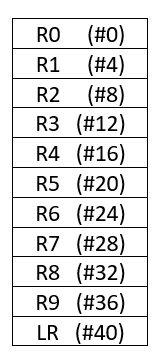


Ilustración 26: Pila con el bloque de activación de la subrutina "candidatos\_propagar\_C" (sp apunta a r0)

#### 5.5.3.2 Recorrer fila

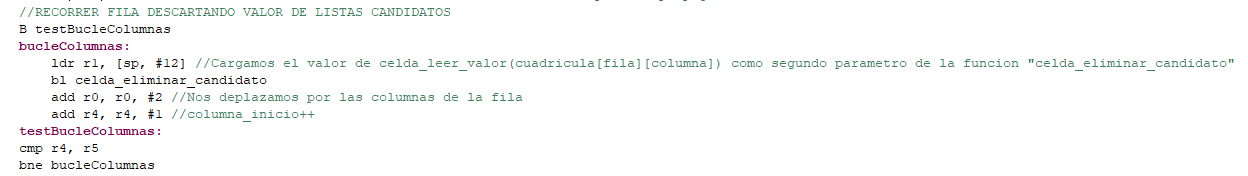


Ilustración 27. Recorrer fila subrutina "candidatos\_actualizar\_arm"

En esta parte del código se ejecuta un bucle para recorrer todas las filas. Dentro de él se carga en r1 el valor de la cuadricula desde la pila (la dirección de la cuadricula ya la tenemos en r0) y llamamos al método “celda\_eliminar\_candidato”.

#### 5.5.3.3 Recorrer columna

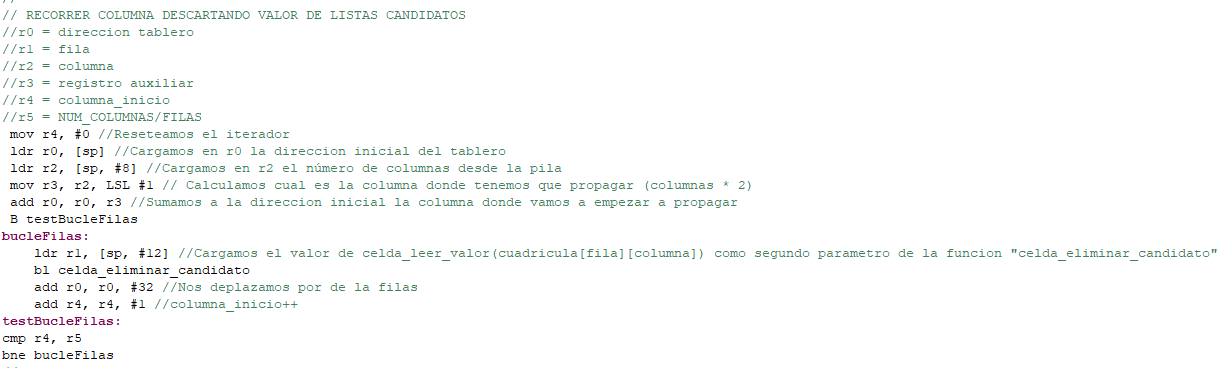


Ilustración 28. Recorrer columnas subrutina "candidatos\_actualizar\_arm"

Esta parte del código es prácticamente igual que la parte anterior salvo que ahora iteramos de 32 bits en 32 bits ya que nos desplazamos en las filas y no en las columnas.

#### 5.5.3.4 Recorrer región

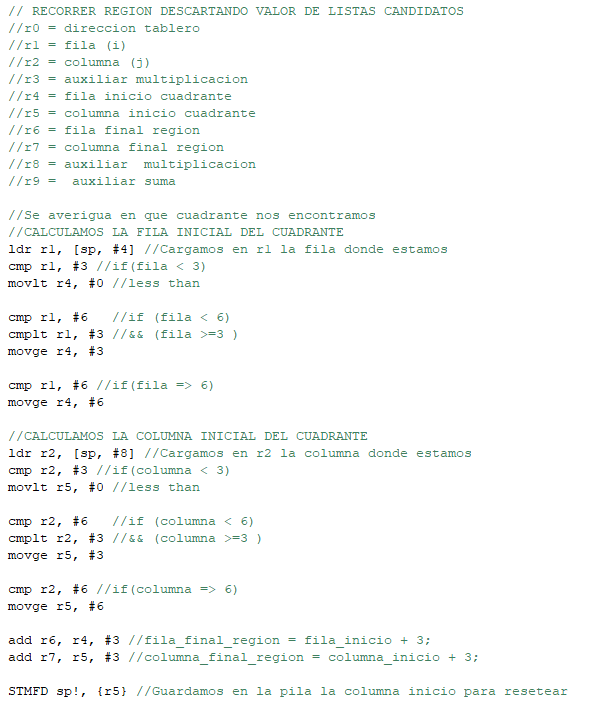


Ilustración 29. Calculamos la región a recorrer en "candidatos\_propagar\_arm"

Igual que en el caso de recorrer la región implementado en C, primero debemos calcular en que región nos encontramos. Una vez calculada la región, sumamos 3 tanto a las filas como a las columnas para marcar el final de región. También guardamos en la pila el valor de la columna inicial para resetearla cuando lleguemos al final del bucle anidado. Por consiguiente, la pila quedaría de la siguiente manera:

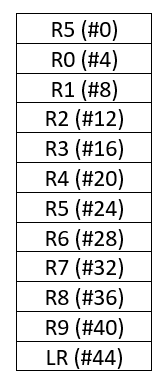


Ilustración 30: Pila tras apilar el valor de columna\_inicio (sp apunta a r5)

Una vez calculada la región a recorrer, ejecutamos el siguiente bucle con otro bucle anidado para recorrerla:

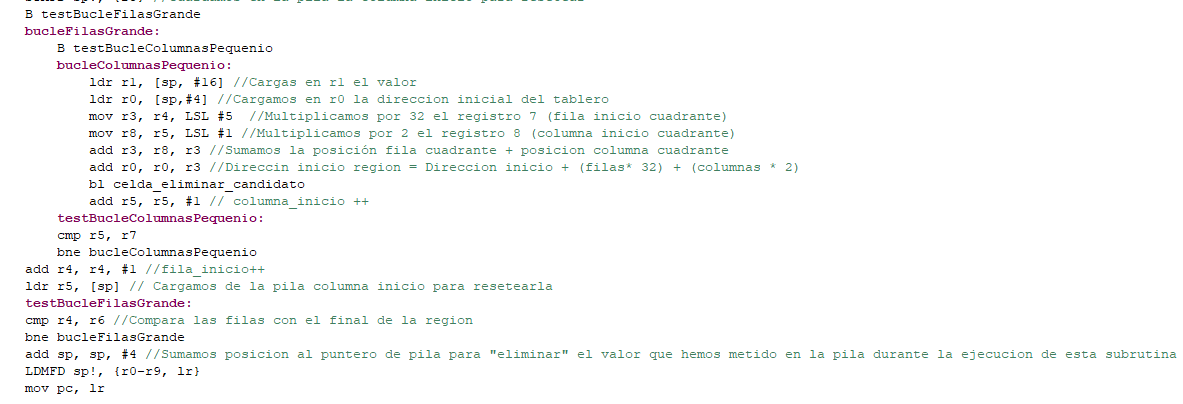


Ilustración 31. Bucle para recorrer la región "candidatos\_propagar\_arm"

En él hacemos lo mismo que en los bucles anteriores, guardamos en r1 el valor que queremos eliminar de los candidatos de la región, después calculamos la dirección de memoria y la guardamos en r0 y, por último, llamamos a “celda\_eliminar\_candidato”.

### 5.5.4 Candidatos\_propagar\_all

La diferencia entre “candidatos\_propagar\_arm” y “candidatos\_propagar\_all” es que en esta última se ejecuta “candidatos\_actualizar\_arm” y “candidatos\_propagar\_arm” en la misma subrutina por lo que no necesitamos guardar dos veces en la pila el bloque de activación. Esto hace que el programa sea un poco más optimo. Por el contrario, debemos usar prácticamente todos los registros.

Como sería tontería volver a poner todo el código ya que hace exactamente lo mismo y está comentado línea por línea en el proyecto, hemos pensado que sería mejor comentar las pequeñas diferencias entre ambos:



Ilustración 32. Bloque de activación "candidatos\_propagar\_all"

La pila quedaría de la siguiente manera:

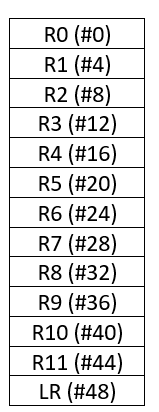


Ilustración 33: Bloque de activación "candidatos\_propagar\_all" guardado en la pila (sp apunta a r0)

Al recorrer las filas, columnas y región nos damos cuenta de que ya no necesitamos cargar de la pila las filas y las columnas ya que siempre hemos tenido ese valor en los registros r5 y r6. Por consiguiente, podemos eliminar las siguientes instrucciones:



Ilustración 34. Instrucción eliminada en recorrer fila



Ilustración 35. Instrucción eliminada en recorrer columna





Ilustración 36. Instrucciones eliminadas en recorrer región

Sin embargo, hemos tenido que utilizar más registros a la hora de recorrer la región ya que los registros que en el método “candidatos\_propagar\_arm” guardaban los límites de dicha región están utilizados por la parte de código que simula “candidatos\_actualizar\_arm”. Estos registros eran r6 y r7 y los hemos tenido que cambiar por r10 y r11:



Ilustración 37. Candidatos\_propagar\_arm



Ilustración 38. Candidatos\_propagar\_all

Por último, recalcar que debemos devolver el número de celdas vacías. Por ello desapilamos el valor de r0 que hemos puesto en la pila, cargamos el valor de “celdas\_vacias” en r0 (valor almacenado en r4) y hacemos el pop de los demás registros de la pila para reestablecer valores y volver a la instrucción que llamó a esta subrutina.

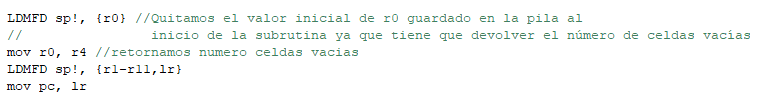


Ilustración 39. Final subrutina "candidatos\_propagar\_all"

## 5.6 Medidas de rendimiento

|  |  |  |
| --- | --- | --- |
| Configuración | Número de instrucciones | Tiempo (milisegundos) |
| C\_C | 43.808 | 4,46 |
| C\_ARM | 33.037 | 3,47 |
| ARM\_C | 42.789 | 4,4 |
| ARM\_ARM | 82.529 | 8,03 |
| ARM\_ALL | 32.130 | 3,45 |

Como podemos observar, salvo por la configuración ARM-ARM, el código implementado en ARM es mucho más eficiente en cuanto a número de instrucciones y de tiempo de ejecución.

# 6. Automatización de la solución

Para poder verificar que las 5 configuraciones realizadas generan la misma salida se ha llevado a cabo la función “cuadricula\_candidatos\_verificar”, mostrada en la ilustración 40. Dicha función compara las diferentes configuraciones con la cuadrícula solución, y devuelve valor 0 si las cuadrículas son iguales y valor 1 en el caso contrario.

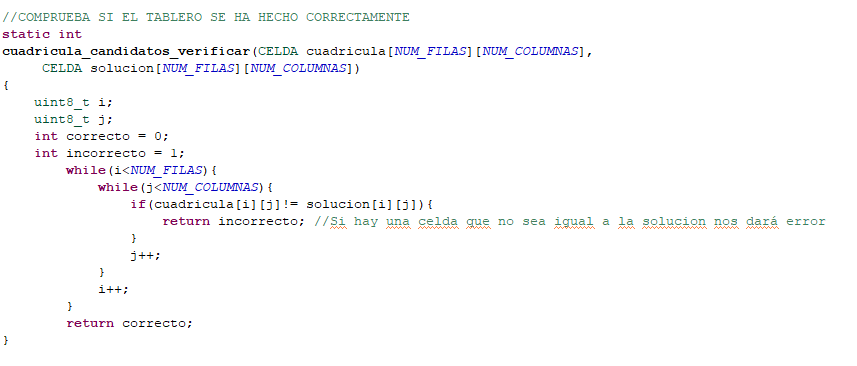


Ilustración 40. Función "cuadricula\_candidatos\_verificar"

En el método sudoku9x9 ejecutamos todas las variantes y para verificar que se ejecutan correctamente llamamos a la función de la ilustración 41 pasando la cuadricula resultante de cada variante y la cuadricula solución (esta cuadrícula viene dada por el profesor). Guardamos el resultado de esa verificación en una estructura que después recorremos para verificar que todas las cuadriculas son correctas.

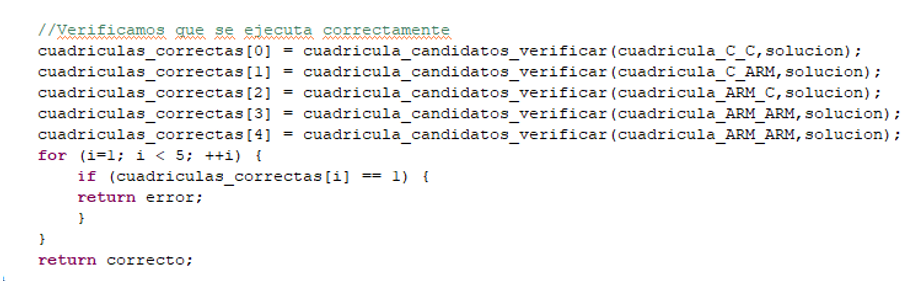


Ilustración 41: Automatización de la solución

# 7. Errores

1. Función propagar

Cuando revisamos la práctica nos dimos cuenta de que leemos el valor de la celda tanto en actualizar como en propagar por lo que vimos que era ineficiente y se podría optimizar. Por consiguiente, decidimos cambiar el esqueleto del código que nos da el profesor para añadir el parámetro “valor” en las llamadas de propagar tanto en C como en ARM:

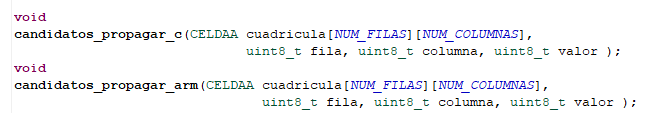


Ilustración 42. Declaración métodos propagar en "sudoku\_2021.h"

De esta forma obtenemos el valor y no necesitamos hacer las siguientes instrucciones al llamar a la subrutina “candidatos\_propagar\_arm”.

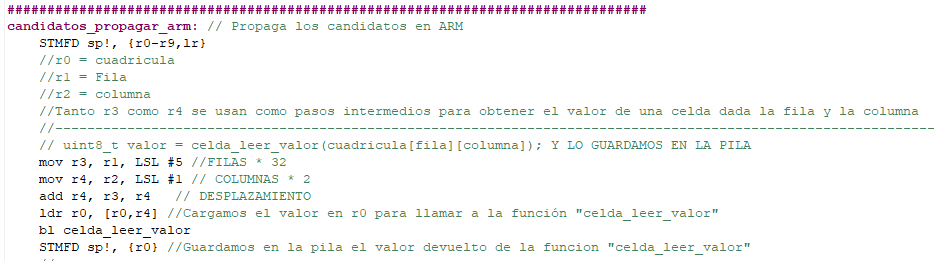


Ilustración 43. Inicio “candidatos\_propagar\_arm” antes de poner el cuarto parámetro

De esta forma nos ahorramos 7 instrucciones cada vez que llamamos a propagar.

En el caso de C nos eliminamos la siguiente instrucción cada vez que lo ejecutamos:

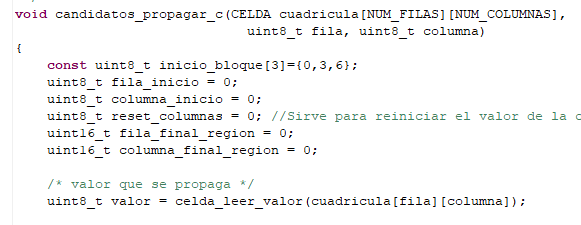


Ilustración 44. Inicio “candidatos\_propagar\_c” antes de poner el cuarto parámetro

1. Main

A la hora de cargar la dirección de cualquier cuadrícula desde ARM nos salía un error ya que dichas cuadrículas estaban definidas en el fichero “tableros.h” y eran estáticas. Para solucionar este problema creamos un método llamado main que será llamado desde ARM y dentro de él iniciamos el programa.

La llamada quedará de la siguiente manera:

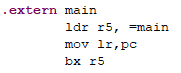


Ilustración 45. Inicio progama ARM

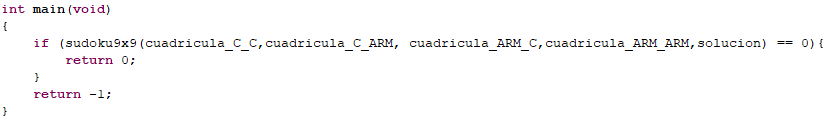


Ilustración 46. Método main del fichero “sudoku\_2021.c”

1. Static

Mediante avanzábamos con la práctica nos dimos cuenta de que los métodos definidos en un fichero diferente a donde los queríamos llamar y que estaban definidos bajo la etiqueta “static” solo podían llamarse desde ese mismo fichero. Esto nos impedía poder llamar a los métodos “celda\_leer\_valor” y “celda\_eliminar\_candidato” por lo que tuvimos que eliminar dicha etiqueta.

1. Global y extern

Al crear un método, tanto en ARM como en C, no podía ser llamado desde el otro lenguaje. Si hacíamos un método en ARM y queríamos ejecutarlo desde C debíamos poner en la cabecera del fichero de ARM la etiqueta “.global” junto con el nombre de la subrutina que deseábamos exportar.

Esto mismo ocurría con los métodos definidos en C que queríamos utilizar en ARM. En este caso debíamos poner la etiqueta “.extern” junto con el nombre del método que queríamos usar en el fichero ARM.

# 8. Conclusión

Esta práctica permite recapacitar sobre la eficacia de un código implementado en ARM frente a un código implementado en C a la vez que nos familiariza con el entorno Eclipse y la interoperabilidad de C-ARM.

# 9. Bibliografía

1º Apuntes de la asignatura Arquitectura y organización de computadores I

2º Apuntes de la asignatura Arquitectura y organización de computadores II

3º Prentice Hall.(1996). *ARM architecture reference manual*