Memoria Práctica 1

Desarrollo de código para el procesador ARM

|  |
| --- |
| Fernando Peña Bes, 756012 |
| Pedro José Pérez García, 756642 |
| Proyecto Hardware  Universidad de Zaragoza, 28 de octubre de 2019 |

Índice

[1. Resumen ejecutivo (una cara como máximo) 3](#_Toc21781870)

[2. Introducción 5](#_Toc21781871)

[2.1 Descripción del juego 5](#_Toc21781872)

[2.2 Entorno de trabajo 6](#_Toc21781873)

[3. Objetivos 6](#_Toc21781874)

[4. Metodología 7](#_Toc21781875)

[Esquema 7](#_Toc21781876)

[Paso 1: Estudiar la documentación 7](#_Toc21781877)

[Paso 2: Estudiar el proyecto ejemplo 7](#_Toc21781878)

[Paso 3: Estudiar y depurar el código inicial del juego 8](#_Toc21781879)

[Paso 4: Realizar patron\_volteo\_arm\_c() en ensamblador ARM 10](#_Toc21781880)

[4.4.1. Ensamblador creado por el compilador 10](#_Toc21781881)

[4.4.2. Ensamblador optimizado 11](#_Toc21781882)

[Paso 5: Realizar una nueva función patron\_volteo\_arm\_arm() 13](#_Toc21781883)

[4.5.1 Ensamblador creado por el compilador 13](#_Toc21781884)

[4.5.2. Ensamblador optimizado 14](#_Toc21781885)

[Paso 6: Verificación automática y comparación de los resultados 15](#_Toc21781886)

[Paso 7: Medidas de tiempo 15](#_Toc21781887)

[Paso 8: Medidas de rendimiento 15](#_Toc21781888)

[Paso 9: Optimizaciones del compilador 15](#_Toc21781889)

[5. Resultados 15](#_Toc21781890)

[6. Conclusiones 15](#_Toc21781891)

[3. Marcos de pila 16](#_Toc21781892)

[4. Código fuente comentado 16](#_Toc21781893)

[1. Descripción de las optimizaciones realizadas al código ensamblador 16](#_Toc21781894)

[1. Resultados de la comparación entre las distintas versiones de las funciones 16](#_Toc21781895)

[2. Descripción de los problemas encontrados en la realización de la práctica y sus soluciones 16](#_Toc21781896)

[3. Conclusiones 16](#_Toc21781897)

[8. Referencias 16](#_Toc21781898)

# 1. Resumen ejecutivo (una cara como máximo)

El objetivo de esta práctica ha sido optimizar el rendimiento de las funciones más costosas computacionalmente de una implementación en C del juego “reversi”, facilitada por los profesores de la asignatura. El código optimizado se ha desarrollado para un procesador ARM y se ha ejecutado sobre una placa de desarrollo real (Embest S3CEV40), usando el entorno de desarrollo Eclipse y el compilador gcc.

Las dos funciones en las que se ha centrado la práctica han sido: patron\_volteo() y   
ficha\_valida() (incluidas en el fichero reversi8\_2019.c). Ambas están estrechamente relacionadas, ya que ficha\_valida() es una función de apoyo a patron\_volteo() y no es utilizada en ninguna otra parte del código del juego. Primero se optimizó únicamente   
patrón\_volteo(), reescribiéndola en ensamblador y manteniendo las llamadas a   
ficha\_valida(). Después, se escribió una única subrutina en ensamblador con la funcionalidad de ambas funciones. Como estas rutinas se debían integrar con el código C original, fue muy importante respetar el estándar AATPCS (ARM Application Procedure Call Standard) al programarlas.

Como parte de la optimización, se realizó un conjunto de pruebas y se tomaron medidas temporales y espaciales. Estas pruebas después se compararon con el código ensamblador generado por el compilador, usando diferentes niveles de optimización.

Para tomar las medidas temporales, se programaron y usaron los temporizadores internos de la placa, mediante las librerías definidas en timer2.c y timer2.h

Los resultados fueron bastante satisfactorios…

2. Introducción

3. Objetivos

Por el momento, como el trabajo es incremental a lo largo de las diferentes entregas, se pretende lograr ejecutar el juego internamente en la memoria de la placa, y en posteriores entregas se irá avanzando en el diseño hacia el juego final que se pretende conseguir.

4. Metodología

Describe los pasos realizados para llegar a los resultados. Todas aquellas decisiones de diseño tomadas en el proceso deben incluirse.

Explicar el esquema del proyecto (ficheros y funciones que lo componen), número total de líneas de código y horas de dedicación. Así como el código desarrollado y comentado.

5. Resultados

Presentar los resultados, explicarlos y analizarlos

6. Conclusiones

Es lo último que se lee, es muy importante, hay que remarcar los mensajes que queremos que el lector reciba. Por ejemplo, si estamos evaluando un producto podemos enfatizar sus puntos fuertes y sus puntos débiles, y señalar posibilidades de mejora.

Normalmente al final se incluyen referencias, bibliografía, índice de expresiones técnicas y anexos.

Importante respetar los requisitos de extensión de la memoria y todos en general.

# 2. Introducción

La práctica recrea una situación en la que una empresa quiere lanzar un sistema, ejecutado sobre un procesador ARM7, que juegue al reversi contra una persona. Por el momento tienen una versión beta del programa en C, pero no están contentos con el tiempo de ejecución. Para ello piden acelerar la función más crítica del juego: patron\_volteo(). El programa debe correr sobre una placa Embest S3CEDV40.

Para empezar, se estudió el código proporcionado y se observaron las funciones que debíamos optimizar. Después se comenzó a trabajar con el entrono de desarrollo y se planteó como realizar las optimizaciones pedidas.

## 2.1 Descripción del juego

Reversi (también llamado Othello o Yang) es un juego muy conocido, principalmente porque, aunque es muy sencillo jugar, es muy complicado dominarlo.

En el reversi, se juega en un tablero de 8 filas por 8 columnas y 64 fichas idénticas, redondas, blancas por una cara y negras por la otra. A un jugador se le asigna un color y se dice que lleva las fichas de ese color, lo mismo para el adversario con el otro color.

Al inicio de la partida, se colocan cuatro fichas en el tablero tal como se ve en el diagrama de la Figura 1.

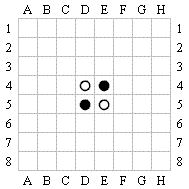


Figura 1 – Tablero de reversi inicial

Empezando por quien lleva las fichas negras los jugadores deben hacer un movimiento por turno, a menos que no puedan hacer ninguno, pasando en ese caso el turno al jugador contrario. El movimiento consiste en colocar una ficha de forma que queden una o varias fichas del color contrario entre la ficha colocada y otras fichas del jugador que mueve. A todas las fichas flanqueadas del adversario se les dan la vuelta, para que pasen a tener el color del jugador que ha colocado la ficha.

Las fichas flanqueadas deben formar líneas continuas rectas (diagonales u ortogonales) de fichas del mismo color entre dos fichas del color contrario (una de ellas debe ser la recién colocada y otra estar presente anteriormente). En el siguiente ejemplo (Figura 2) juegan primero las fichas negras y después las blancas, se puede ver que fichas se voltean cuando los jugadores realizan cada movimiento.

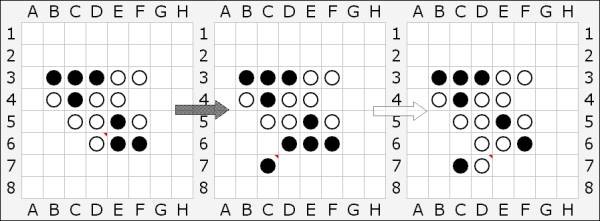


Figura 2 - Dos movimientos sobre el tablero de reversi

La partida finaliza cuando ningún jugador puede mover (normalmente cuando el tablero está lleno de fichas) y gana quien en ese momento tenga sobre el tablero más fichas mostrando su color.

## 2.2 Entorno de trabajo

Se ha utilizado Eclipse, junto con las herramientas gcc de compilación cruzada. La placa (Embest S3CEV40) se ha utilizado con un soporte especial a través del puerto JTAG que permite la ejecución paso a paso y acceso en tiempo real al estado del procesador y memoria.

Para poder depurar el código fuera del laboratorio, instalamos el entorno en nuestros ordenadores personales y usamos el plug-in de eclipse para simular procesadores ARM7TDMI.

# 3. Objetivos

La finalidad de esta práctica es comprender como trabaja un compilador para ARM y conseguir optimizar partes del código programándolas directamente en ensamblado, profundizando en la interacción C / Ensamblador.Para ello es necesario comprender el estado arquitectónico de la máquina (contenido de registros y memoria) y entender la finalidad y el funcionamiento de las ABI (Aplication Binary Interface), en este caso, el estándar AATPCS.

También es importante aprender a interactuar con el entorno de programación y ser capaces de interactuar con una placa real, ejecutando y deputando código sobre ella. Además, se tendrá que aprender a gestionar la entrada/salida con dispositivos básicos, asignando valores a los registros correspondientes. Y por último, saber gestionar el tiempo de trabajo del proyecto correctamente en función de la disponibilidad de acceso a la placa de desarrollo.

# 4. Metodología

## Esquema

En primer lugar, se incluye un esquema del proyecto con los diferentes ficheros utilizados.

PH\_Practica1

├── 8led.c

├── 8led.h

├── button.c

├── button.h

├── led.c

├── led.h

├── main.c

├── README.md

├── reversi8\_2019.c

├── timer.c

├── timer.h

├── subrutinasP1.asm

├── timer2.c

├── timer2.h

├── init\_b\_2018.asm

└── common

├── 44b.h

├── 44binit.asm

├── 44blib.c

├── 44blib.h

├── def.h

├── ev40boot.cs

├── ld\_script.ld

├── Memcfg.a

├── option.a

└── option.h

Los fichero subrayados son los que hemos realizado nosotros durante la práctica. El juego se encuentra en el fichero reversi8\_2019.c. La carpeta common incluye todos los ficheros necesarios para inicializar la placa.

## Pasos 1 y 2: Estudiar la documentación y el proyecto de ejemplo

Para empezar, se estudió el funcionamiento del sistema y la documentación proporcionada. Una vez que nos familiarizamos con el entorno y la placa, estudiamos el proyecto de ejemplo del contador timer0.

Después se estudió el funcionamiento de las funciones resaltadas

Después se programó un temporizador para poder evaluar el rendimiento y se estudiaron las opciones del compilador para mejorar las prestaciones del código generado

Explicación del timer0 y como se controlan las interrupciones (ejecutar sobre la placa)

El timer0 se gestionará utilizando interrupciones IRQ. Las rutinas de tratamiento de interrupción se desarrollarán en C siguiendo la misma estructura que en el proyecto que os damos. El compilador se ocupará de algunos de los detalles a bajo nivel, pero debéis ser capaces de entender el código ensamblador que genera el compilador y explicar qué hace en cada paso. De nuevo, la descripción del marco de pila utilizado deberá́ aparecer convenientemente explicado en la memoria.

El timer0 utiliza interrupciones IRQ,

## Paso 3: Estudiar y depurar el código inicial del juego

El siguiente paso fue ejecutar y entender el código en C del reversi.

Para poderlo ejecutar se modificó la función Main() del fichero main.c para que llamara la función reversi8(). Se mantuvieron las llamadas a las funciones de inicialización de la placa para que no hubiera problemas a la hora de ejecutar el código.



Código 1 - Función Main() con llamada a reversi8()

La función reversi8() es la que contiene el proceso principal del juego. Utiliza una matriz 8x8 de datos de tipo char para almacenar el tablero. Siempre es el jugador quien realiza el primer movimiento (por lo que siempre le corresponden las fichas negras). El programa espera a que el usuario introduzca una fila y columna y ponga una señal ready a 1 (que indica que se quiere realizar el movimiento). Una vez hecho esto, el programa actualiza el tablero con la ficha del usuario, calcula un movimiento, coloca una ficha blanca, y vuelve a esperar a que el usuario mueva otra vez. No se comprueba que el usuario realice un movimiento correcto según las reglas, sólo que el movimiento de la máquina sea válido. El juego termina cuando no hay más movimientos posibles, entonces se realiza un recuento de los puntos y se almacena en memoria.

Esta primera versión del juego se juega directamente en la memoria de la placa, así que probarlo, se colocó un monitor de memoria en inicio del tablero, y la representación de la memoria se configuró en grupos de 1 byte y colocando 8 grupos por fila. Como cada posición del tablero ocupa 1 byte podemos tener una representación en memoria del tablero en forma matricial (Figura 3).

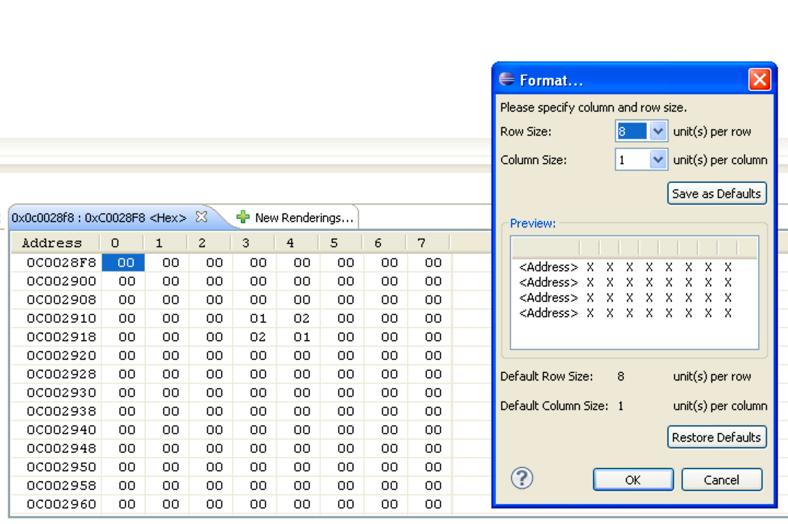


Figura 3 - Monitorización del tablero en memoria

El tablero empieza en la posición 0x0c0028f8 en este caso y se puede ver como están colocadas las 4 fichas iniciales. El valor 0 corresponde a casilla libre, el 1 a casilla con ficha blanca, y el 2 a casilla con ficha negra.

Las coordenadas es también necesario introducirlas a mano en la memoria. La función   
esperar\_mov() espera a que el usuario ponga la variable ready a 1 para añadir la ficha al tablero.

Se observa el vector en el que se guarda el tablero de reversi (tablero) y las variables de control del usuario (fila, columna y ready) son variables globales, aunque deberían ser locales de   
reversi8(), esto se ha hecho así porque al meterlas en la pila el compilador no las pondría juntas, por lo que jugar sería más complicado. De esta forma los cuatro elementos están en posiciones contiguas de memoria y se puede introducir la jugada fácilmente accediendo a las posiciones 0x0c002938 (fila), 0x0c002939 (columna) y 0x0c00293a (ready). Se puede ver en el siguiente ejemplo (Figuras 4 y 5):

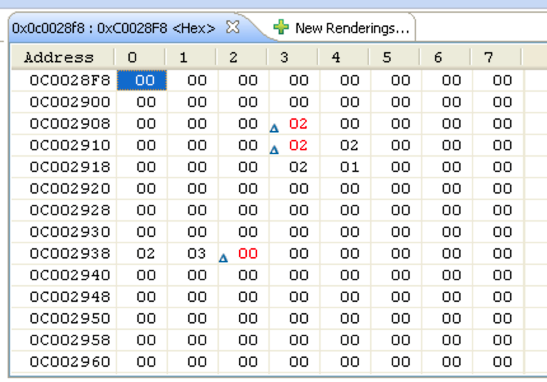


Figura 4 - Actualización tablero

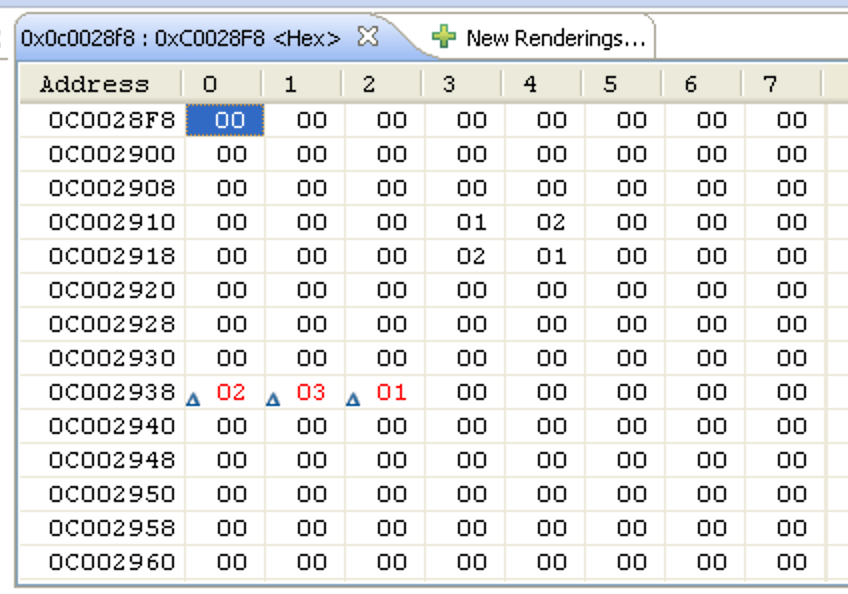


Figura 5 - Introducción movimiento usuario

Notar como se ha dado la vuelta a las fichas blancas correspondientes y como el programa vuelve a poner a 0 la variable ready una vez introducido el movimiento.

Acto seguido el programa introduce una ficha blanca y actualiza las casillas necesarias siguiendo las reglas del juego (Figura 6):

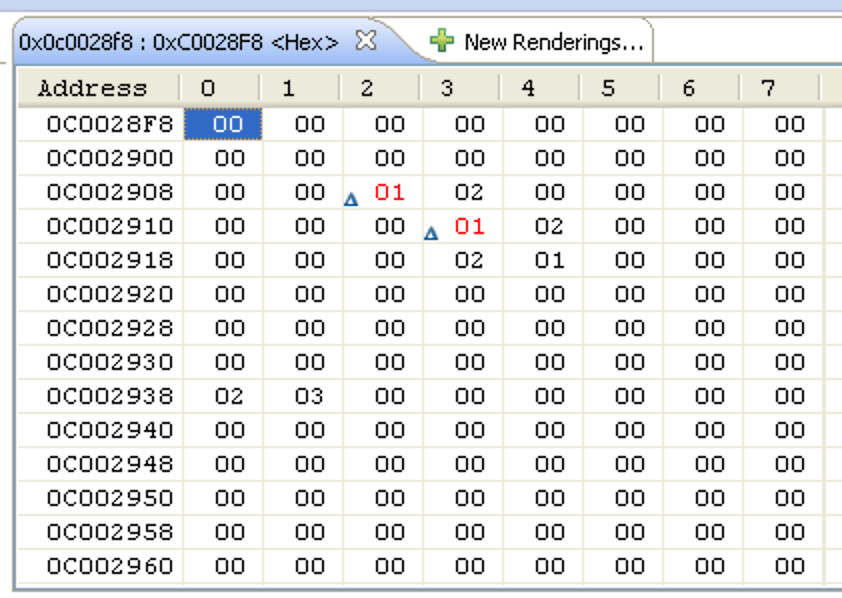


Figura 6 - Movimiento realizado por el programa

Una vez visto el funcionamiento general del programa, se analizaron las funciones   
patron\_volteo() y ficha\_valida(). El código en C de ambas es el siguiente:

|  |
| --- |
| ////////////////////////////////////////////////////////////////////////////////  // Devuelve el contenido de la posición indicadas por la fila y columna actual.  // Además informa si la posición es válida y contiene alguna ficha.  // Esto lo hace por referencia (en \*posicion\_valida)  // Si devuelve un 0 no es válida o está vacia.  **char** **ficha\_valida**(**char** tablero[][*DIM*], **char** f, **char** c, **int** \*posicion\_valida)  {  **char** ficha;  // ficha = tablero[f][c];  // no puede accederse a tablero[f][c]  // ya que algún índice puede ser negativo  **if** ((f < *DIM*) && (f >= 0) && (c < *DIM*) && (c >= 0) && (tablero[f][c] != *CASILLA\_VACIA*))  {  \*posicion\_valida = 1;  ficha = tablero[f][c];  }  **else**  {  \*posicion\_valida = 0;  ficha = *CASILLA\_VACIA*;  }  **return** ficha;  }  ////////////////////////////////////////////////////////////////////////////////  // La función patrón volteo comprueba si hay que actualizar una determinada direccion,  // busca el patrón de volteo (n fichas del rival seguidas de una ficha del jugador actual)  // en una dirección determinada  // SF y SC son las cantidades a sumar para movernos en la dirección que toque  // color indica el color de la pieza que se acaba de colocar  // la función devuelve PATRON\_ENCONTRADO (1) si encuentra patrón y NO\_HAY\_PATRON (0) en caso  // contrario  // FA y CA son la fila y columna a analizar  // longitud es un parámetro por referencia. Sirve para saber la longitud del patrón que se está  // analizando.  // Se usa para saber cuantas fichas habría que voltear  **int** **patron\_volteo**(**char** tablero[][*DIM*], **int** \*longitud, **char** FA, **char** CA, **char** SF, **char** SC,  **char** color)  {  **int** posicion\_valida; // indica si la posición es valida y contiene una ficha de algún  jugador  **char** casilla; // casilla es la casilla que se lee del tablero  FA = FA + SF;  CA = CA + SC;  casilla = ficha\_valida(tablero, FA, CA, &posicion\_valida);  **while** ((posicion\_valida == 1) && (casilla != color))  // mientras la casilla está en el tablero, no está vacía,  // y es del color rival seguimos buscando el patron de volteo  {  FA = FA + SF;  CA = CA + SC;  \*longitud = \*longitud + 1;  casilla = ficha\_valida(tablero, FA, CA, &posicion\_valida);  }  // si la ultima posición era válida y la ficha es del jugador actual,  // entonces hemos encontrado el patrón  **if** ((posicion\_valida == 1) && (casilla == color) && (\*longitud >0))  **return** *PATRON\_ENCONTRADO*; // si hay que voltear una ficha o más hemos encontrado el  // patrón  **else**  **return** *NO\_HAY\_PATRON*; // si no hay que voltear no hay patrón  } |

patron\_volteo() se encarga de comprobar si dada una casilla y una dirección de movimiento, se puede encontrar una serie de fichas en línea a las que se pueda dar la vuelta siguiendo las reglas del juego, es decir, que estén entre la ficha introducida y otra de ese mismo color. Además de devolver si ha encontrado un patrón de volteo, devuelve la longitud del mismo.

El algoritmo consiste en ir avanzando desde la casilla inicial (FA, CA) en la dirección indicada por SF y SC mientras haya fichas en el tablero del color contrario a la ficha colocada. Si justo después de una línea de fichas de ese color hay otra del color de la colocada, se ha encontrado un patrón.

La función ficha\_valida() dada una casilla del tablero devuelve en por el parámetro   
posición\_valida un 1 si la casilla está dentro de los límites del tablero y hay alguna ficha en esa posición. Devuelve 0 en caso contrario. Si la posición es válida, además devuelve el color de la ficha en esa posición (1 o 2), en el caso contrario devuelve CASILLA\_VACIA (0).

## Paso 4: Realizar patron\_volteo\_arm\_c() en ensamblador ARM

### 4.4.1. Ensamblador creado por el compilador

Antes de realizar esta función se estudió el código en ensamblador producido por el compilador sin aplicar optimizaciones (Código 1) de la función patron\_volteo().

Llama mucho la atención que se usan pocos registros, y casi todos los datos que se usan se van guardando y leyendo de memoria, lo que es muy poco eficiente.

La llamada a patrón\_volteo se realiza de la siguiente manera:

|  |
| --- |
| 0c0015fc: ldrb r2, [r11, #-45] ; 0x2d 0c001600: ldrb r3, [r11, #-46] ; 0x2e 0c001604: ldrb r1, [r11, #4] 0c001608: str r1, [sp] 0c00160c: ldrb r1, [r11, #8] 0c001610: str r1, [sp, #4] 0c001614: ldrb r1, [r11, #12] 0c001618: str r1, [sp, #8] 0c00161c: ldr r0, [r11, #-40] ; 0x28 0c001620: ldr r1, [r11, #-44] ; 0x2c 0c001624: bl 0xc0014c8 <patron\_volteo> 0c001628: str r0, [r11, #-16] |

Y el código de la subrutina correspondiente a patron\_volteo es la siguiente:

|  |
| --- |
| patron\_volteo: 0c001328: mov r12, sp 0c00132c: push {r11, r12, lr, pc} 0c001330: sub r11, r12, #4 0c001334: sub sp, sp, #24 0c001338: str r0, [r11, #-24] 0c00133c: str r1, [r11, #-28] 0c001340: strb r2, [r11, #-29] 0c001344: strb r3, [r11, #-30]  205 FA = FA + SF; 0c001348: ldrb r2, [r11, #-29] 0c00134c: ldrb r3, [r11, #4] 0c001350: add r3, r2, r3 0c001354: strb r3, [r11, #-29]  206 CA = CA + SC; 0c001358: ldrb r2, [r11, #-30] 0c00135c: ldrb r3, [r11, #8] 0c001360: add r3, r2, r3 0c001364: strb r3, [r11, #-30]  207 casilla = ficha\_valida(tablero, FA, CA, &posicion\_valida); 0c001368: ldrb r1, [r11, #-29] 0c00136c: ldrb r2, [r11, #-30] 0c001370: sub r3, r11, #20 0c001374: ldr r0, [r11, #-24] 0c001378: bl 0xc001274 <ficha\_valida> 0c00137c: mov r3, r0 0c001380: strb r3, [r11, #-13]  208 while ((posicion\_valida == 1) && (casilla != color)) 0c001384: b 0xc0013d8 <patron\_volteo+176> 0c0013d8: ldr r3, [r11, #-20] 0c0013dc: cmp r3, #1 0c0013e0: bne 0xc0013f4 <patron\_volteo+204> 0c0013e4: ldrb r2, [r11, #-13] 0c0013e8: ldrb r3, [r11, #12] 0c0013ec: cmp r2, r3 0c0013f0: bne 0xc001388 <patron\_volteo+96>  219 if ((posicion\_valida == 1) && (casilla == color) && (\*longitud >0)) 0c0013f4: ldr r3, [r11, #-20] 0c0013f8: cmp r3, #1 0c0013fc: bne 0xc001428 <patron\_volteo+256> 0c001400: ldrb r2, [r11, #-13] 0c001404: ldrb r3, [r11, #12] 0c001408: cmp r2, r3 0c00140c: bne 0xc001428 <patron\_volteo+256> 0c001410: ldr r3, [r11, #-28] 0c001414: ldr r3, [r3] 0c001418: cmp r3, #0 0c00141c: ble 0xc001428 <patron\_volteo+256>  220 return PATRON\_ENCONTRADO; // si hay que voltear una ficha o más  hemos encontrado el patrón 0c001420: mov r3, #1 0c001424: b 0xc00142c <patron\_volteo+260>  222 return NO\_HAY\_PATRON; // si no hay que voltear no hay patrón 0c001428: mov r3, #0  223 } 0c00142c: mov r0, r3 0c001430: sub sp, r11, #12 0c001434: ldm sp, {r11, sp, lr} 0c001438: bx lr |

Código 2 - patron\_volteo() compilado con gcc

### 4.4.2. Ensamblador optimizado

#### Marco de pila

Lo primero que se hizo antes de empezar a programar la subrutina fue definir el marco de pila a utilizar.

Se respetó el estándar AAPCS, que incluye un conjunto de convenios:

* Define el uso de los registros de propósito general.
* Define cómo se utiliza la pila (*full descending*).
* Define la estructura de los datos de la pila, que se utiliza para depurar los programas.
* Define el mecanismo de pasar argumentos y el resultado en una función que debe usarse para hacer visibles externamente las funciones y procedimientos, es decir, que la llamada puede hacerse desde fuera del módulo de programación actual. Una función que sólo se utiliza dentro de un módulo puede omitir el estándar.
* Da soporte al mecanismo de bibliotecas compartidas de ARM, que quiere decir que da soporte al estándar para compartir código para acceder a datos estáticos.

La convención de uso de registros por el estándar es el siguiente (Tabla 1):

|  |  |  |
| --- | --- | --- |
| **Registro** | **Nombre según AAPCS** | **Función según AAPCS** |
| 0 | a1 | Argumento 1 / resultado subrutina / scratch register |
| 1 | a2 | Argumento 2 / scratch register |
| 2 | a3 | Argumento 3/ scratch register |
| 3 | a4 | Argumento 4 / scratch register |
| 4 | v1 | Variable 1 |
| 5 | v2 | Variable 2 |
| 6 | v3 | Variable 3 |
| 7 | v4 | Variable 4 |
| 8 | v5 | Variable 5 |
| 9 | sb/v6 | Static base / Variable 6 |
| 10 | sl/v7 | Stack limit / Variable 7 |
| 11 | fp | Frame pointer |
| 12 | ip | Scratch register / Intra Procedure call scratch Register |
| 13 | sp | Stack pointer |
| 14 | lr | Link register /scratch register |
| 15 | pc | Program counter |

Tabla 1 - Convención AAPCS de uso de registros

Además, el estándar define registros preservados (r4-r11, r13 y r14) y no preservados (r0-r3 y r12). La subrutina tiene que asegurarse que se el valor de los registros preservados es el mismo antes y después de su ejecución, mientras que puede modificar el de los no preservados.

Sin embargo, como este estándar no cubre todas las posibilidades al realizar el marco de pila, se optó por utilizar uno similar al generado por el compilador. De esta forma, las comparaciones de rendimiento serán más justas. El marco de pila es el siguiente (Figura 7):

|  |  |
| --- | --- |
| -36 | poscion\_valida |
| -16 | r4-r10 |
| -12 | fp |
| -8 | sp |
| -4 | lr |
| fp → | pc |
| +4 | SF |
| +8 | SC |
| +12 | color |
|  | Marco de pila  subrutina  anterior |

Figura 7 - Marco de pila patron\_volteo\_arm\_c()

Los 4 primeros parámetros de la función se pasan por r0-r3 y los tres que quedan se pasan en la pila. Además, dentro de la subrutina, se reserva espacio para la posicion\_válida(), no se reserva espacio para casilla, ya que siempre que se vuelva de la llamada a ficha\_valida(), el valor correspondiente a casilla estará en r0 y no hace falta guardarlo en la pila. El resultado de la subrutina se almacena en r0, como dice el estándar, y se asegura que se restaure el valor inicial de los registros preservados antes de retornar al programa principal.

La versión realizada a mano en ensamblador de la función es la siguiente (Código 3):

|  |
| --- |
| .equ DIM, 8  .equ DESPL, 3 @2^DESPL = DIM  .equ CASILLA\_VACIA, 0  .equ NO\_HAY\_PATRON, 0  .equ PATRON\_ENCONTRADO, 1  **patron\_volteo\_arm\_c:**  mov ip, sp  push {r4-r10, fp, ip, lr, pc}  sub fp, ip, #4  sub sp, #4 @ Espacio para posicion\_valida  @ En r0 está @tablero  @ En r1, @longitud  @ En r2, FA  @ En r3, CA  mov r4, r0 @ r4 = @tablero  mov r5, r1 @ r5 = @longitud  mov r6, r2 @ r6 = FA  mov r7, r3 @ r7 = CA  @ Cargar parámetros pasados por la pila  ldrsb r8,[fp,#4] @ r8 = SF  ldrsb r9,[fp,#8] @ r9 = SC  ldr r10,[r5] @ Cargo longitud en r10    **WHILE\_C:** add r6,r6,r8 @ FA = FA + SF  add r7, r7, r9 @ CA = CA + SC  @ Llamada a la función ficha\_valida  mov r0, r4 @ r0 = @tablero  mov r1, r6 @ r1 = FA  mov r2, r7 @ r2 = CA  sub r3, fp, #36 @ r3 = @ de posicion\_valida  bl ficha\_valida @ En r0 tenemos casilla y en  fp-36 posicion\_valida  @ ((posicion\_valida == 1) && (casilla != color))  ldr r1,[fp,#-36] @ r1 = posicion\_valida  cmp r1, #1 @ Comprobar si la posicion es valida  bne FIN\_WHILE\_C  ldr r3,[fp, #12] @ r3 = color  cmp r0, r3 @ Comparar casilla con color  beq FIN\_WHILE\_C  @ \*longitud = \*longitud + 1  add r10,r10,#1 @ Incrementar longitud  b WHILE\_C  **FIN\_WHILE\_C:** @ ((posicion\_valida == 1) && (casilla == color) && (longitud > 0))  cmp r1,#1 @ posicion\_valida == 1  cmpeq r0, r3 @ casilla == color  cmpeq r10, #0 @ longitud > 0  movhi r0, #PATRON\_ENCONTRADO  movls r0, #NO\_HAY\_PATRON  str r10,[r5] @ Almacenar longitud calculada  ldmdb fp,{r4-r10, fp, sp, pc} |

Código 3 - patron\_volteo\_arm\_c()

El funcionamiento de la subrutina, como se puede ver en los comentarios, es una traducción directa del código en C. Sin embargo, se han usado todos los registros necesarios para evitar leer de memoria el mismo valor varias veces. Además, se han usado condiciones predicadas en algunos casos para mejorar el rendimiento evitando saltos. También se han usado instrucciones de transferencia de datos múltiples (PUSH y LDMDB) para trabajar con la pila, ya que son mas eficientes que instrucciones de acceso a memoria separadas.

## Paso 5: Realizar una nueva función patron\_volteo\_arm\_arm()

### 4.5.1 Ensamblador creado por el compilador

|  |
| --- |
| 0c00155c: ldrb r1, [r11, #-29] 0c001560: ldrb r2, [r11, #-30] 0c001564: sub r3, r11, #20 0c001568: ldr r0, [r11, #-24] 0c00156c: bl 0xc001414 <ficha\_valida> 0c001570: mov r3, r0 0c001574: strb r3, [r11, #-13] |

|  |
| --- |
| ficha\_valida: 0c001414: mov r12, sp 0c001418: push {r11, r12, lr, pc} 0c00141c: sub r11, r12, #4 0c001420: sub sp, sp, #24 0c001424: str r0, [r11, #-24] 0c001428: str r3, [r11, #-32] 0c00142c: mov r3, r1 0c001430: strb r3, [r11, #-25] 0c001434: mov r3, r2 0c001438: strb r3, [r11, #-26] 181 if ((f < DIM) && (f >= 0) && (c < DIM) && (c >= 0) && (tablero[f][c] != CASILLA\_VACIA)) 0c00143c: ldrb r3, [r11, #-25] 0c001440: cmp r3, #7 0c001444: bhi 0xc0014a0 <ficha\_valida+140> 0c001448: ldrb r3, [r11, #-26] 0c00144c: cmp r3, #7 0c001450: bhi 0xc0014a0 <ficha\_valida+140> 0c001454: ldrb r3, [r11, #-25] 0c001458: lsl r3, r3, #3 0c00145c: ldr r2, [r11, #-24] 0c001460: add r2, r2, r3 0c001464: ldrb r3, [r11, #-26] 0c001468: ldrb r3, [r2, r3] 0c00146c: cmp r3, #0 0c001470: beq 0xc0014a0 <ficha\_valida+140> 183 \*posicion\_valida = 1; 0c001474: ldr r3, [r11, #-32] 0c001478: mov r2, #1 0c00147c: str r2, [r3] 184 ficha = tablero[f][c]; 0c001480: ldrb r3, [r11, #-25] 0c001484: lsl r3, r3, #3 0c001488: ldr r2, [r11, #-24] 0c00148c: add r2, r2, r3 0c001490: ldrb r3, [r11, #-26] 0c001494: ldrb r3, [r2, r3] 0c001498: strb r3, [r11, #-13] 0c00149c: b 0xc0014b4 <ficha\_valida+160> 188 \*posicion\_valida = 0; 0c0014a0: ldr r3, [r11, #-32] 0c0014a4: mov r2, #0 0c0014a8: str r2, [r3] 189 ficha = CASILLA\_VACIA; 0c0014ac: mov r3, #0 0c0014b0: strb r3, [r11, #-13] 191 return ficha; 0c0014b4: ldrb r3, [r11, #-13] 192 } 0c0014b8: mov r0, r3 0c0014bc: sub sp, r11, #12 0c0014c0: ldm sp, {r11, sp, lr} 0c0014c4: bx lr 205 { |

#### Marco de pila

La función ficha\_valida() estaba programada de forma independiente para mejorar la claridad, pero como sólo se llama desde patron\_volteo(), se puede incrustar dentro del código de esta función. Esto es lo que se ha hecho en este apartado. Al hacerlo, además de poder optimizar la versión generada por el compilador, se ahorra el coste de llamar a la subrutina. La nueva subrutina en ensamblador que contiene la funcionalidad de ambas funciones es   
patron\_volteo\_arm\_arm().

El marco de pila utilizado para esta subrutina es similar al anterior, pero se ahorra tener que almacenar en la pila posición\_valida, ya que su valor se puede guardar en un registro directamente (Figura 8).

|  |  |
| --- | --- |
| -16 | r4-r10 |
| -12 | fp |
| -8 | sp |
| -4 | lr |
| fp → | pc |
| +4 | SF |
| +8 | SC |
| +12 | color |
|  | Marco de pila  subrutina  anterior |

Figura 8 - Marco de pila patron\_volteo\_arm\_c()

### 4.5.2. Ensamblador optimizado

El código realizado se muestra a continuación (Código 4):

|  |
| --- |
|  |

Código 4 - patron\_volteo\_arm\_arm()

// Comentar funcionamiento, se ha respetado el algoritmo

## Paso 6: Verificación automática y comparación de los resultados

Para probar que las subrutinas creadas tuvieran el mismo comportamiento que la función inicial en C, se creó una nueva función patron\_volteo\_test(), que verifica que las diferentes implementaciones de patron\_volteo generan la misma salida. La salida incluye el valor devuelto por la función y el valor de longitud, que es usado como parámetro de entrada-salida.

La función recibe los mismos parámetros que patrón volteo, y ejecuta las tres distintas implementaciones. Si los resultados no coinciden, la función indica el error quedándose en un bucle infinito. La implementación es la siguiente.

|  |
| --- |
| **patron\_volteo\_test**(**char** tablero[][*DIM*], **int** \*longitud, **char** FA, **char** CA, **char** SF, **char** SC, **char** color)  {  **int** patron\_c\_c, patron\_arm\_c, patron\_arm\_arm;  // Ejecutar patron\_volteo  patron\_c\_c = patron\_volteo(tablero, longitud, FA, CA, SF, SC, color);  **int** longitud\_c\_c = \*longitud;  \*longitud = 0;  // Ejecutar patron\_volteo\_arm\_c  patron\_arm\_c = patron\_volteo\_arm\_c(tablero, longitud, FA, CA, SF, SC, color);  **int** longitud\_arm\_c = \*longitud;  \*longitud = 0;  // Ejecutar patron\_volteo\_arm\_arm  patron\_arm\_arm = patron\_volteo\_arm\_arm(tablero, longitud, FA, CA, SF, SC, color);  **int** longitud\_arm\_arm = \*longitud;  // Comprobar que los resultados de todas las funciones sean iguales  **if** (patron\_c\_c != patron\_arm\_c || patron\_c\_c != patron\_arm\_arm) {  **while** (1);  }  **if** (longitud\_c\_c != longitud\_arm\_c || longitud\_c\_c != longitud\_arm\_arm) {  **while**(1);  }  **return** patron\_c\_c;  } |

Código 5 - Código patron\_volteo\_test()

## Paso 7: Medidas de tiempo

Para poder realizar mediciones de tiempo, tal y como se especifica en el enunciado de la práctica, la siguiente tarea ha sido desarrollar la biblioteca que permita controlar el timer2, con la máxima precisión posible y generando el mínimo número de interrupciones, devolviendo las cuentas de tiempo en microsegundos.

Teniendo las especificaciones del enunciado en cuenta, se ha llegado a la conclusión de que la mejor forma de evitar sobrecargar el timer de interrupciones es inicializar el valor de la cuenta a 65535, valor máximo posible del registro de 16 bits, el preescalado se ha configurado a 0, y el divisor del MUX se ha ajustado a ½.

Esto es, que el procesador nos está dando una interrupción cada 2 ciclos, por tanto, el contador decrementa cada 0.03125 microsegundos. Ahora, si tenemos que hacer toda la cuenta regresiva, tenemos 65535 × 0.03125, se produce una interrupción cada 2047.96 microsegundos (en el código final la variable destinada a este cálculo tiene un valor de 2048).

Cada vez que se desee leer el valor de la cuenta del timer2, se debe leer la variable con el número de interrupciones que se incrementa en la interrupción, y multiplicarla por 2048, aparte de sumarle el número de ticks que se llevan de la siguiente cuenta divididos por 32, ya que coincide que 64 ticks de reloj (32 con el divisor del MUX) son un microsegundo.

La fórmula final para calcular tiempos es:

|  |
| --- |
| **timer2\_num\_int \* PERIOD\_INT + (rTCNTB2 - rTCNTO2) / CYCLES\_EACH\_MICROSEC** //interrupciones \* 2048 + cuenta\_ticks\_actual / ciclos\_reloj\_por\_cada\_microsegundo |

Código 6 - Obtención del valor en microsegundos de la cuenta actual, usado en timer2\_leer()

#### Marco de pila

A continuación se detalla el marco de la pila cada vez que llega una nueva interrupción del timer, como se puede ver es el marco propio de una IRQ que llega en un momento arbitrario de la ejecución de cualquier otro fragmento del programa, por lo que tiene que asegurar que la ejecución de esta subrutina no afecta en nada al resto de la ejecución principal cuando vuelva.

|  |  |
| --- | --- |
| -20 | r2 |
| -16 | r3 |
| -12 | r11 |
| -8 | r12 (sp) |
| -4 | lr |
| fp → | pc |
| +4 | r12 |

Figura 9 - Marco de pila timer2\_ISR()

Es una pila con una estructura diferente por el hecho de ser una interrupción, donde apila primero r12 porque luego guarda en ese registro el valor de sp para volver a apilarlo, posiblemente como medida de seguridad para no perderlo, luego apila pc, lr, r12 con el valor del sp resultante de apilar por primera vez r12 y r2 y r3, que usará para aumentar el contador de interrupciones.

Por último, tras apilar todo, da a fp el valor de sp - 4, que queda apuntando a pc en la pila.

#### Implementación de las funciones en lenguaje C

El código de la biblioteca está pensado para no interferir con cualquier otro periférico que se haya podido conectar con la placa, de forma que en los registros de configuración no se fuerzan los valores deseados desde las funciones de inicialización y se utilizan operaciones como el AND o el OR de C.

|  |
| --- |
| void timer2\_inicializar(void)  {  /\* Configuración controlador de interrupciones \*/  rINTMOD &= 0xFFFFF7FF; //Configura la línea del timer2 como IRQ  rINTCON &= 0x1; // Habilita int. vectorizadas y linea IRQ, dejando FIQ como estuviera  rINTMSK &= ~(BIT\_TIMER2); // habilitamos en vector de mascaras de interrupción el Timer0 (bits 26 y 13, BIT\_GLOBAL y BIT\_TIMER0 están definidos en 44b.h)  /\* Establece la rutina de servicio para TIMER0 \*/  pISR\_TIMER2 = (unsigned) timer2\_ISR;  /\* Configura el Timer2 \*/  rTCFG0 &= 0xFFFF00FF; // ajusta el preescalado a 0  rTCFG1 &= 0xFFFFF0FF; // selecciona la entrada del mux que proporciona el reloj. La 00 corresponde a un divisor de 1/2.  } |

Código 7 - timer2\_inicializar()

También se han adoptado algunas decisiones en el diseño del código, por ejemplo, solo se inicializa timer2\_num\_int a 0 en timer2\_empezar() y no en la función de inicialización, ya que no es necesario, o en timer2\_parar no se repite código y solo se hace parar la cuenta, junto con una llamada a timer2\_leer()

En cuanto a optimizaciones o camios posibles en el diseño se podría haber utilizado un desplazamiento de bits para las operaciones descritas en la Fórmula 1 más arriba, ya que algunos valores son potencias de 2, pero se decidió no hacerlo en caso de que algún valor pudiera cambiar en el futuro, y así se evitan posibles modificaciones futuras del código.   
También, durante la corrección de la práctica se advirtió un problema respecto a la forma de leer los valores del timer, ya que si entre dos lecturas cambian los valores de la variable que cuenta el número de interrupciones o el de los ticks se pueden dar valores erróneos. Para solventar esto, se medirán dos veces los valores, y si se detecta que algo significativo ha cambiado, se usarán los valores más nuevos. El código final para leer queda:

|  |
| --- |
| unsigned int timer2\_leer(void)  {  unsigned int num\_int\_1 = timer2\_num\_int;  unsigned int num\_int\_2 = timer2\_num\_int;  //Con esto, evitamos posibles incrementos no deseados en timer2\_num\_int  if(num\_int\_2 > num\_int\_1)  {  return num\_int2 \* PERIOD\_INT + (rTCNTB2 - rTCNTO2) / CYCLES\_EACH\_MICROSEC;    }  else  {  return num\_int1 \* PERIOD\_INT + (rTCNTB2 - rTCNTO2) / CYCLES\_EACH\_MICROSEC;    }  } |

Código 8 - Versión final de timer2\_leer()

Finalmente, para probar su correcto funcionamiento y su precisión, se preparó una batería de pruebas automática, que toma medidas 10 veces para 1ms, 10ms, 1 segundo y 10 segundos, almacena cada medida en una componente de un vector y calcula la media, para poder evaluar los retardos.

|  |
| --- |
| timer2\_inicializar(); //Para la prueba se inicializa el timer2  int i;  unsigned int suma\_1ms = 0;  unsigned int medidas\_1ms[10];  for(i=0; i < 10; i = i + 1)  {  timer2\_empezar();  unsigned int t\_0 = timer2\_leer();  Delay(10);  unsigned int t\_1 = timer2\_parar();  medidas\_1ms[i] = t\_1 - t\_0;  suma\_1ms += medidas\_1ms[i];  }  unsigned int media\_1ms = suma\_1ms / 10; |

Código 9 - Fragmento del código para validar timer2

Tras evaluar los resultados con el visor de memoria de Eclipse, los resultados promedio de las 10 mediciones para cada Delay fueron:

* Para 1 milisegundo, 1022 microsegundos.
* Para 10 milisegundos, 10169 microsegundos.
* Para 1 segundo, 1017015 microsegundos.
* Para 10 segundos, 10170125 microsegundos.

Por tanto, se concluye que las mediciones son suficientemente acertadas, considerando que ya se nos advierte que la función Delay() utilizada para las pruebas no está perfectamente calibrada y que el desajuste es demasiado pequeño.

El código completo de timer2.c y timer2.h, así como de pruebas\_timer2.c se puede encontrar entre los materiales de la entrega del código.

## Paso 8: Medidas de rendimiento

Una vez comprobado que las dos implementaciones de patron\_volteo() eran correctas, se midieron los tiempos de ejecución de cada una de ellas para cada una de las opciones de optimización ofrecidas por el compilador (-O0, -O1, -O2, -O3 y -Os). También se midió el tiempo para la versión arm\_arm optimizada descrita en el apartado 4.5.2.

Para realizar las medidas, se usó el timer2, cuyos detalles se han explicado en la sección anterior. Se midió el tiempo de ejecución de patron\_volteo() al colocar en el tablero inicial una ficha negra en la casilla (2,3). Como patrón volteo se ejecuta 8 veces cada vez que se coloca una ficha (una para cada dirección) se apuntó el tiempo de cada una de las ejecuciones, ya que generalmente varía el número de instrucciones ejecutadas dependiendo de la dirección. La quinta iteración es aquella en la que se encuentra la ficha en estas pruebas, por lo que tardará más.

Los resultados obtenidos fueron los que se detallan a continuación (en microsegundos):

Figura 10

Figura 11

Figura 12

Figura 13

De los datos obtenidos con estas mediciones, se puede apreciar que la ejecución de la versión ARM\_C es la más costosa de media, seguramente por cuestiones de reordenado o de organización de memoria del compilador, siendo la versión más rápida de todas la de C optimizada con -O3, con un coste de 6 microsegundos en su mejor caso, y de 8 en la quinta iteración, cuando encuentran la ficha.

Figura 14

Por otro lado, la versión ARM optimizada parece ser la que presenta un mejor comportamiento en general, seguida por la versión ARM-ARM sin optimizar.

## Paso 9: Optimizaciones del compilador

Figura 15

También se intentaron comprobar y analizar las diferentes técnicas y estrategias que sigue el compilador con el código, en función de los diferentes niveles de optimización que ofrece. Para ello se han considerado únicamente los códigos de patron\_volteo() y patron\_volteo\_arm\_c().

Para llevar a cabo el análisis, se ha compilado con los diferentes niveles de optimización un Main sencillo que llama a la función correspondiente, con un tablero predefinido, y se ha comparado su desensamblado con el de la Figura //poner\_numero , en caso de ser la versión en C, y el de la figura //poner\_numero, si se trata de la versión híbrida entre C y ensamblador.

A continuación se ilustran los resultados de la compilación, y también se comentan las técnicas de optimización observadas:

|  |
| --- |
| **131 patron\_c\_c = patron\_volteo(tablero1, &longitud, 2, 3, -1, 0, 2);**  **0c0011b8: sub r3, r11, #28**  **0c0011bc: mov r2, #255 ; 0xff**  **0c0011c0: str r2, [sp]**  **0c0011c4: mov r2, #0**  **0c0011c8: str r2, [sp, #4]**  **0c0011cc: mov r2, #2**  **0c0011d0: str r2, [sp, #8]**  **0c0011d4: ldr r0, [pc, #56] ; 0xc001214 <Main+132>**  **0c0011d8: mov r1, r3**  **0c0011dc: mov r2, #2**  **0c0011e0: mov r3, #3**  **0c0011e4: bl 0xc00107c <patron\_volteo>**  **0c0011e8: mov r3, r0**  **0c0011ec: str r3, [r11, #-16]**  **132 patron\_c\_c += 1;**  **0c0011f0: ldr r3, [r11, #-16]**  **0c0011f4: add r3, r3, #1**  **0c0011f8: str r3, [r11, #-16]**  **133 patron\_c\_c -=1;**  **0c0011fc: ldr r3, [r11, #-16]**  **0c001200: sub r3, r3, #1**  **0c001204: str r3, [r11, #-16]**  **134 }**  **0c001208: sub sp, r11, #12**  **0c00120c: ldm sp, {r11, sp, lr}**  **0c001210: bx lr**  **0c001214: stceq 8, cr1, [r0], {128} ; 0x80** |

Código 10 – Código patrón volteo en C, optimizado con -O1

# 5. Resultados

# 6. Conclusiones

# 3. Marcos de pila

# 4. Código fuente comentado

Además, cada función debe incluir una cabecera en la que se explique como funciona, qué parámetros recibe y donde los recibe y para qué usa cada registro (por ejemplo, en el registro 4 se guarda el puntero a la primera matriz...)

# 1. Descripción de las optimizaciones realizadas al código ensamblador

# Resultados de la comparación entre las distintas versiones de las funciones

# Descripción de los problemas encontrados en la realización de la práctica y sus soluciones

# Conclusiones

# 8. Referencias

GuiaEntorno.pdf

EntradaSalida.pdf

P2-ec.pdf

Material de apoyo para la asignatura de Arquitectura y Organización de Computadores 1.

[Figura 1 – Tablero de reversi inicial 5](#_Toc21733568)

[Figura 2 - Dos movimientos sobre el tablero de reversi 6](#_Toc21733569)

[Figura 3 - Monitorización del tablero en memoria 9](#_Toc21733570)

[Figura 5 - Actualización tablero 9](https://unizares-my.sharepoint.com/personal/756642_unizar_es/Documents/PH/Memoria%20P1%20PH.docx#_Toc21733571)

[Figura 4 - Introducción movimiento usuario 9](https://unizares-my.sharepoint.com/personal/756642_unizar_es/Documents/PH/Memoria%20P1%20PH.docx#_Toc21733572)

[Figura 6 - Movimiento realizado por el programa 10](#_Toc21733573)

[Código 1 - Función Main() con llamada a reversi8() 8](#_Toc21733577)