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 2

2. Introducción 3

2.1 Descripción del juego 3

2.2 Entorno de trabajo 4

3. Objetivos 4

4. Metodología 5

Esquema 5

Pasos 1 y 2: Estudiar la documentación y el proyecto de ejemplo 5

Paso 3: Estudiar y depurar el código inicial del juego 6

Paso 4: Realizar `patron_volteo_arm_c()` en ensamblador ARM 9

4.4.1. Ensamblador creado por el compilador 9

4.4.2. Ensamblador optimizado 11

Paso 5: Realizar una nueva función `patron_volteo_arm_arm()` 14

4.5.1 Ensamblador creado por el compilador 14

4.5.2 Ensamblador optimizado 16

4.5.2. Ensamblador optimizado 16

Paso 6: Verificación automática y comparación de los resultados 18

Paso 7: Medidas de tiempo 19

Paso 8: Medidas de rendimiento 21

Paso 9: Optimizaciones del compilador 23

Apartado opcional 1 26

Apartado opcional 2 28

5. Problemas encontrados y soluciones 28

6. Conclusiones 28

7. Referencias 29

1. Resumen ejecutivo

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 `patron_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. Se aseguro que todas las funciones programadas respetaran el algoritmo original de la versión en C.

Una vez programadas las funciones, se realizó un conjunto de pruebas automáticas para garantizar su correcto funcionamiento 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. Además se creó una nueva versión de `patron_volteo()` en ensamblador optimizada, ligeramente alejada del algoritmo original, para mejorar la eficiencia.

Para tomar las medidas temporales, se programaron y usaron los temporizadores internos de la placa, específicamente `timer2`. La librería que contiene las funciones para trabajar con él se encuentra en los ficheros `timer2.c` y `timer2.h`

Los resultados fueron, en general, los esperados. La función en ensamblador optimizada era siempre la que menos ocupaba en memoria y la más rápida.

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.

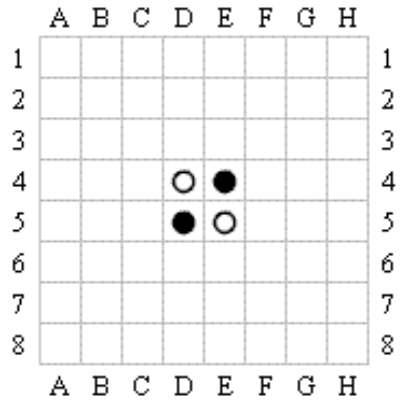

Figura 1 – Tablero de reversi inicial

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.

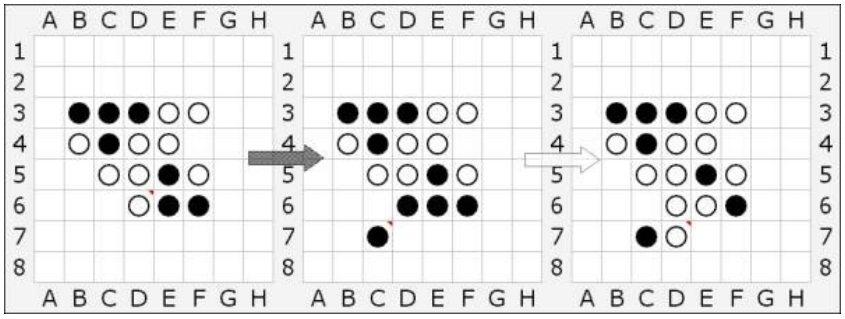

Figura 2 - Dos movimientos sobre el tablero de reversi

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
├── LEEME_P1.txt
├── reversi8_2019.c
├── reversi8_2019.h
├── timer.c
├── timer.h
├── main_medidas_optimizaciones.c
├── pruebas_timer2.c
├── patron_volteo_arm_c.asm
├── patron_volteo_arm_arm.asm
├── patron_volteo_arm_arm_opt.asm
├── timer2.c
├── timer2.h
└── common
      ├── 44b.h
      ├── 44binit.asm
      ├── 44blib.c
      ├── 44blib.h
      ├── def.h
      ├── ev40boot.cs
      ├── ld_script.ld
      ├── Memcfg.a
      ├── option.a
      └── option.h
```

Los ficheros 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`.

El `timer0` se gestiona utilizando interrupciones IRQ. Tiene un contador interno que va decrementando de acuerdo con un valor de preescalado y un valor de divisor, cuando llega a un límite establecido, lanza una excepción, y cuando llega a 0, se reinicia la cuenta.

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.

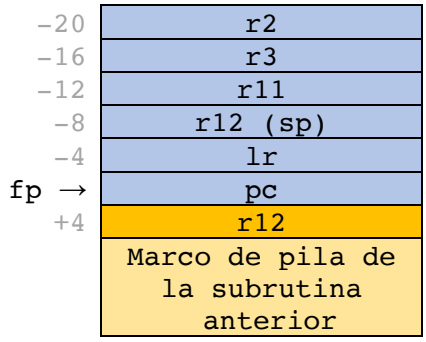

Figura 3 - Marco de pila `timer2_ISR()`

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.

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(void)` 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.

```
void Main(void)
{
	/* Inicializa controladores */
	sys_init();		// Inicializacion de la placa, interrupciones y puertos
	timer_init();		// Inicializacion del temporizador
	Eint4567_init();	// inicializamos los pulsadores. Cada vez que se pulse
				   se verá reflejado en el 8led
	D8Led_init();		// inicializamos el 8led

	reversi8();
}
```

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 4).

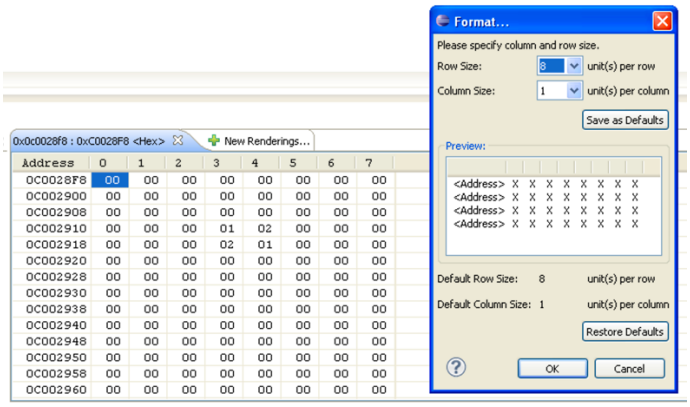

Figura 4 - Monitorización del tablero en memoria

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 (Figura 6Figura 5):

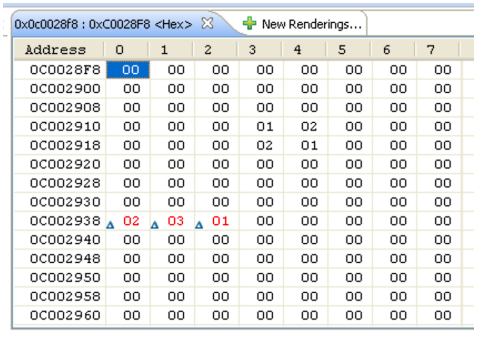

Figura 6 - Introducción movimiento usuario

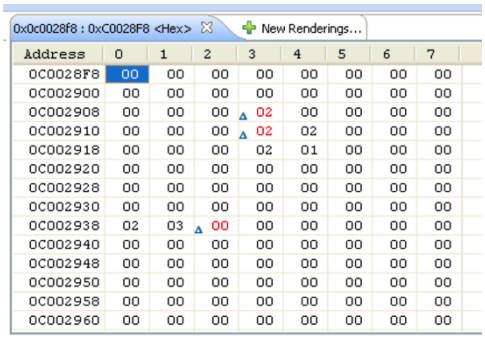

Figura 5 - Actualización tablero

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 7):

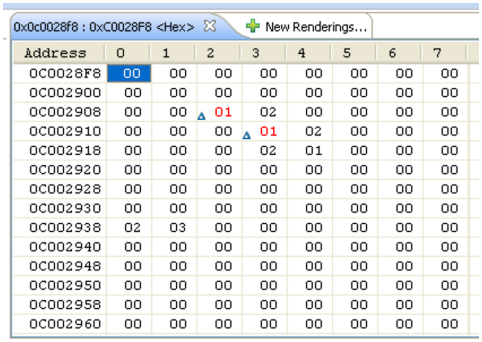

Figura 7 - Movimiento realizado por el programa

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


}
```

Código 2 - Código de `patron_volteo()` original en C

`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 `patron_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]
```

Código 3 - Llamada a `patron_volteo()` en la versión C - C

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 4 - `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 8):

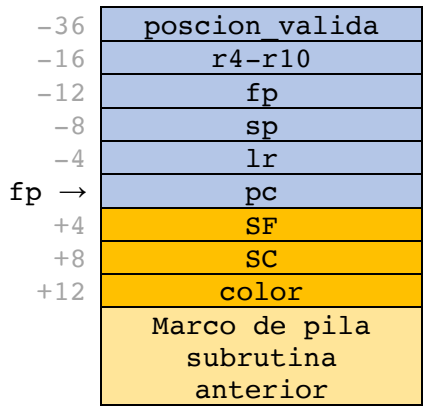

Figura 8 - Marco de pila `patron_volteo_arm_c()`

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_valida()`, 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 5):

```
@ Devuelve si ha encontrado un patrón de volteo y la longitud analizada
en el parámetro *longitud
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
                        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
                        movne r0, #NO_HAY_PATRON
@ Si posicion_valida != 0 o casilla != color, no hay patron. Devolver
no hay patron
                        bne FIN_C
                        cmpeq r10, #0           @ longitud > 0
                        movhi r0, #PATRON_ENCONTRADO @ devolver patrón
encontrado
                        movls r0, #NO_HAY_PATRON @ devolver patrón no
encontrado
FIN_C:                  str   r10,[r5]          @ Almacenar longitud
calculada
                        ldmdb fp,{r4-r10, fp, sp, pc}
```

Código 5 – **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

En el apartado anterior únicamente se implementó la función `patron_ volteo()`, pero este se pedía crear una subrutina única en ARM que tuviera la función de `patron_volteo()` y `ficha_valida()`. Para comenzar, se examinó el código generado por el compilador para esta última función.

La llamada a la función y el código generado por el compilador son los siguientes. Como en el caso anterior, la gestión de la memoria y los registros podría ser mejor, ya que no se aprovechan todos los registros disponibles y se realizarán múltiples accesos a memoria y saltos.

```
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]
```

Código 6 - Llamada a `ficha_valida()` en versión C - C

```
          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] != CA-
SILLA_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       {
```

Código 7 - Ensamblador generado por `gcc` para `ficha_valida()` en la versión C - C

4.5.2 Ensamblador optimizado

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 9).

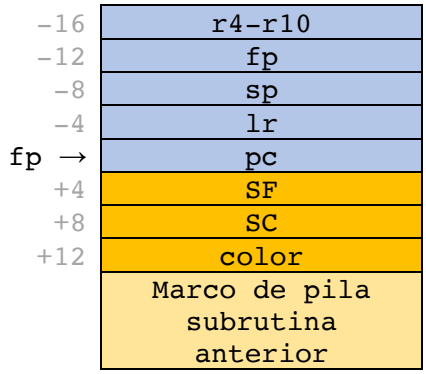

Figura 9 - Marco de pila `patron_volteo_arm_c()`

4.5.2. Ensamblador optimizado

El código realizado se muestra a continuación (Código 8):

```
@ Devuelve si ha encontrado un patrón de volteo y la longitud anali-
zada en el parámetro *longitud
patron_volteo_arm_arm:
                        mov ip, sp
                        push {r4-r10, fp, ip, lr, pc}
                        sub fp, ip, #4
                        @ r0=@tablero
                        @ r1=@longitud
                        @ r2=FA
                        @ r3=CA
                        ldrsb r4,[fp,#4]  @r4 = SF
                        ldrsb r5,[fp,#8]  @r5 = SC
                        ldrb r6,[fp,#12]  @r6 = color

                        ldr r8, [r1] @ r8 = longitud inicial

                        @ FA = FA + SF
WHILE_ARM:          add r2, r2, r4
                        @ CA = CA + SC
                        add r3, r3, r5
```

```
                        @ Llamada a ficha válida
                        @ r7=tablero[FA][CA] (casilla),
                          r9=posicion_valida
                        @ FA < DIM && FA >= 0 && CA < DIM && CA >= 0
&& (tablero[FA][CA] != CASILLA_VACIA)
                        cmp r2, #DIM
                        bge NO_VALIDA           @ FA >= DIM
                        cmp r2, #0
                        blt NO_VALIDA           @ FA < 0
                        cmp r3, #DIM
                        bge NO_VALIDA           @ CA >= DIM
                        cmp r3, #0
                        blt NO_VALIDA           @ CA < 0
                        @ r10=@ de la posicion en el tablero a anali-
zar
                        add r10, r0, r2, LSL #DESPL @añadir fila
                        add r10, r10, r3  @añadir columna
                        ldrb r7, [r10]          @ r7=tablero[FA][CA]
                        cmp r7, #CASILLA_VACIA
                        mov r9, #1
                        bne WHILE_COND_ARM            @r7 != CASI-
LLA_VACIA

NO_VALIDA:        mov r9, #0              @posicion_valida=0
                        mov r7, #CASILLA_VACIA

WHILE_COND_ARM: @(posicion_valida == 1) && (casilla != color)
                        cmp r9, #1
                        bne IF_ARM
                        cmp r7, r6
                        beq IF_ARM        @ tablero[FA][CA] == color
                        @ sumar 1 a longitud
                        add r8, r8, #1
                        b WHILE_ARM


IF_ARM:           cmp r9, #1   @posicion_valida==1
                        cmpeq r7, r6 @casilla==color
                        movne r0, #NO_HAY_PATRON @ Si posicion_valida
!= 0 o casilla != color, no hay patron. Devolver no hay patron
                        bne FIN_ARM
                        cmpeq r8, #0
                        movhi r0, #PATRON_ENCONTRADO @ Devolver pa-
tron encontrado
                        movls r0, #NO_HAY_PATRON @ Devolver no hay
patron
FIN_ARM:          str r8, [r1]      @ Almacenar longitud calculada
                        ldmdb fp,{r4-r10, fp, sp, pc}
```

Código 8 - `patron_volteo_arm_arm()`

El funcionamiento es muy similar al código original, se ha mantenido el algoritmo al máximo para que las posteriores medidas de rendimiento sean justas.

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 9 - Código `patron_volteo_test()`

En este paso, para probar la función, se sustituyeron todas las llamadas a `patron_volteo()` en el juego (dos) por esta función y se jugó probando diferentes combinaciones manualmente. Se probó a colocar una ficha en los lados, en las esquinas, a formar un patrón de volteo y a intentar engañar al programa con patrones incorrectos.

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 10 - Obtención del valor en microsegundos de la cuenta actual, usado en `timer2_leer()`

Marco de pila

El marco de pila de `timer2` es el generado por el compilador, y es similar al del `timer0` descrito en el paso 2.

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)
{
	/* Configuraion controlador de interrupciones */
	rINTMOD &= 0xFFFFF7FF; // Configura la línea del timer2 como IRQ
	rINTCON &= 0x1; // Habilita int. vectorizadas y la linea IRQ, dejando
	                   FIQ como estuviera
	rINTMSK &= ~(BIT_TIMER2); // habilitamos en vector de mascaras de
	                             interrupcion 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 11 - `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 cambios 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)
{
	return timer2_num_int * PERIOD_INT +
			(rTCNTB2 - rTCNTO2) / CYCLES_EACH_MICROSEC;
//Si queremos optimizar, como la multiplicación es por 2048, se pueden mover
los bits 16 lugares a al izquierda y en la división, al ser por 32, se pueden
mover 5 a la derecha.
}}
```

Código 12 - 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 13 - 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 y tamaño en instrucciones de de cada una de ellas para cada una de y se compararon con el código inicial.

Para medir los tiempos de ejecución se usó el `timer2` programado en el apartado anterior La prueba consistió en medir el tiempo de ejecución de `patron_volteo()` al colocar en el tablero inicial una ficha negra en la casilla (2,3).

Cada vez que se coloca una ficha, `patron_volteo()` se ejecuta 8 veces cada vez que se coloca una ficha (una para cada dirección) en busca de un patrón de volteo. Es el caso que hemos planteado la ficha se encuentra rodeada por casillas vacías menos por debajo. Cuando la ficha tiene casillas vacías o con ficha del mismo color contiguas, `patron_volteo()` únicamente realiza una iteración ya que puede descartar que haya un patrón volteo al examinar la primera casilla. Sin embargo, cuando la primera casilla examinada es tiene una ficha del otro color, recorre la dirección en busca del patrón. En nuestro caso, como se refleja en la Figura 10, cuando se busca el patrón volteo hacia abajo se encuentra un patrón de longitud 1, por lo que esa ejecución tarda más.

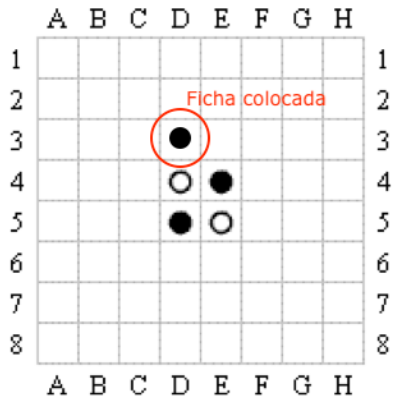

Figura 10

Se realizaron tres mediciones de la prueba usando las tres implementaciones de `patron_volteo()`:

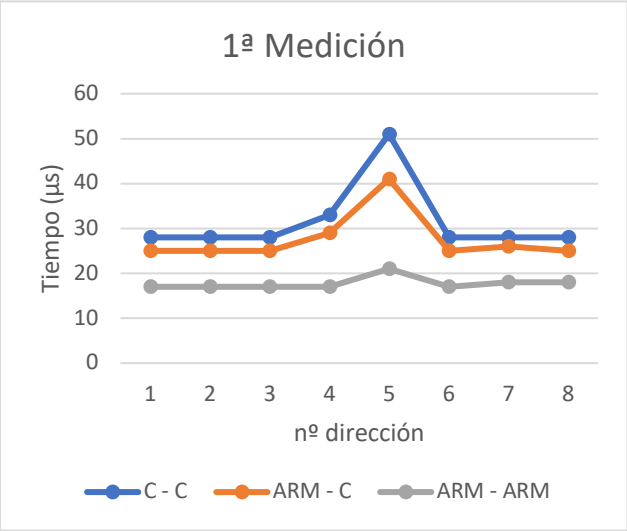

Figura 12

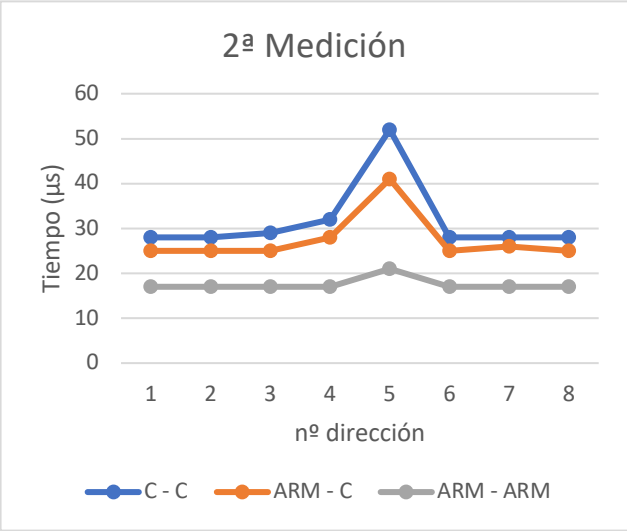

Figura 11

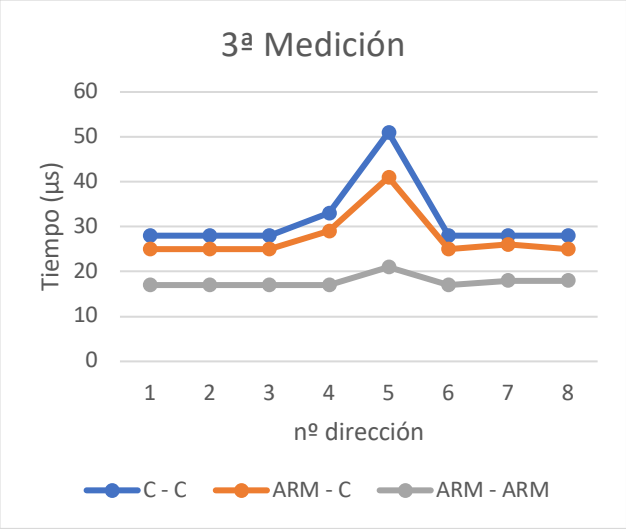

Figura 13

Las tres mediciones presentan resultados similares y se puede ver como la versión ARM-ARM es la más rápida, tal y como se espera. Destaca que las versiones C-C y ARM-C tienen tiempos muy similares, seguramente por el sobrecoste de llamar a la subrutina `ficha_valida()`.

En cuanto a la evaluación del tamaño en código, se calculó a partir de la posición inicial y final del los desensamblados de las funciones proporcionados por el debugger de Eclipse ARM. Cada instrucción ocupa 4 bytes, así que, para obtener el tamaño del código en bytes, se multiplicó el número de instrucciones por 4.

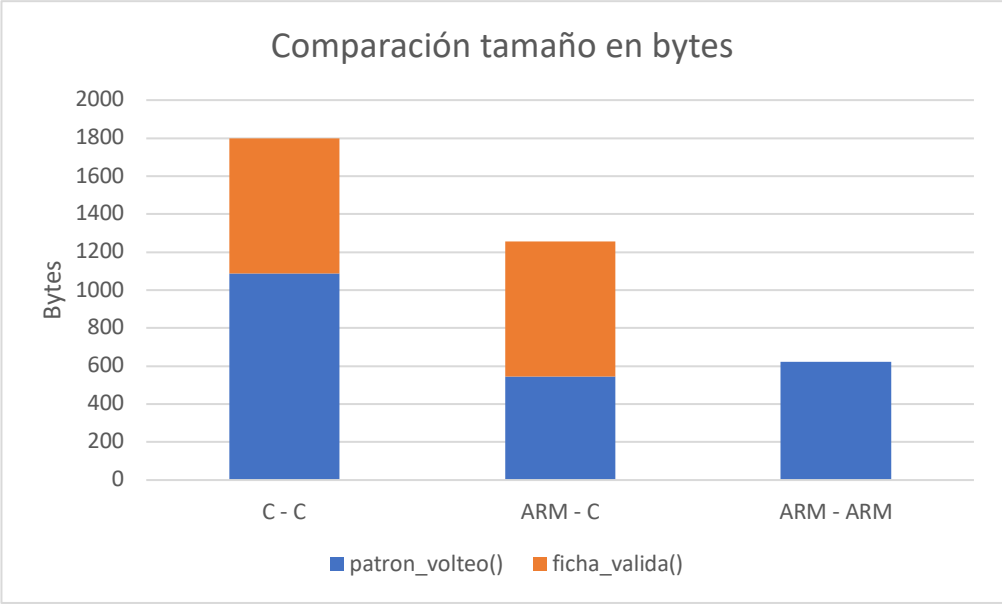

Figura 14

Se puede ver como la versión ARM-ARM es la que menos ocupa.

Paso 9: Optimizaciones del compilador

Una vez comprobado que las dos implementaciones de `patron_volteo()` eran correctas, se midieron los tiempos de ejecución y tamaño en instrucciones 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 ha analizado la versión ARM-ARM optimizada que se describirá en el apartado opcional 1.

Según la referencia de `gcc`, se describen las optimizaciones de la siguiente manera:

- **-01** El compilador intenta reducir el tamaño del código y el tiempo de ejecución, pero sin realizar ninguna optimización que tome demasiado tiempo de compilación
- **-02** El compilador realiza casi todas las optimizaciones soportadas que no manteniendo el balance espacio-velocidad. Comparado con -O1, esta incrementa el tiempo de compilación y el rendimiento del código generado.
- **-O3** Optimiza todavía más que -O2, añadiendo optimizaciones adicionales.
- **-Os** Activa todas las optimizaciones

Primero se midieron los tiempos de las 4 implementaciones de `patron_volteo()` usando las diferentes opciones de optimización. La prueba realizada fue la misma que en el apartado anterior, midiendo los tiempos de ejecución al coloca la una ficha negra en la posición (2,3) en el tablero inicial.

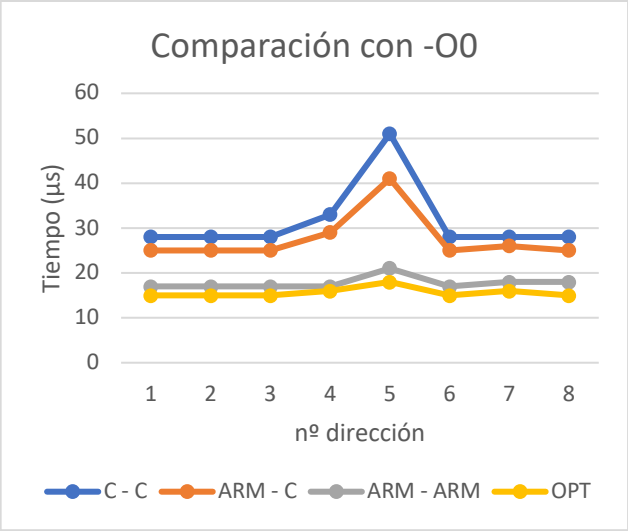

Figura 16

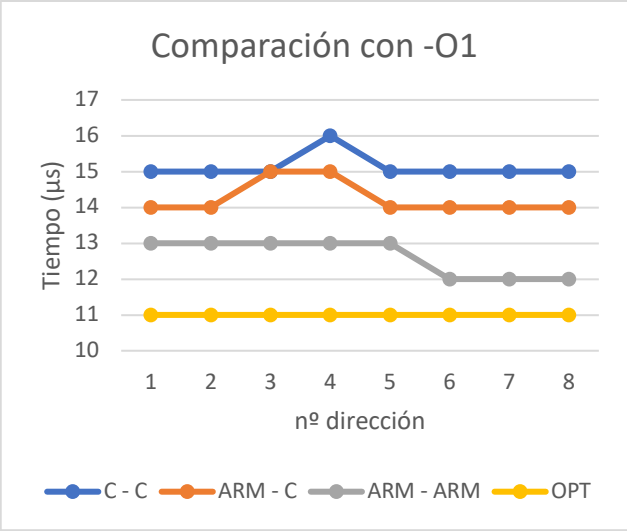

Figura 15

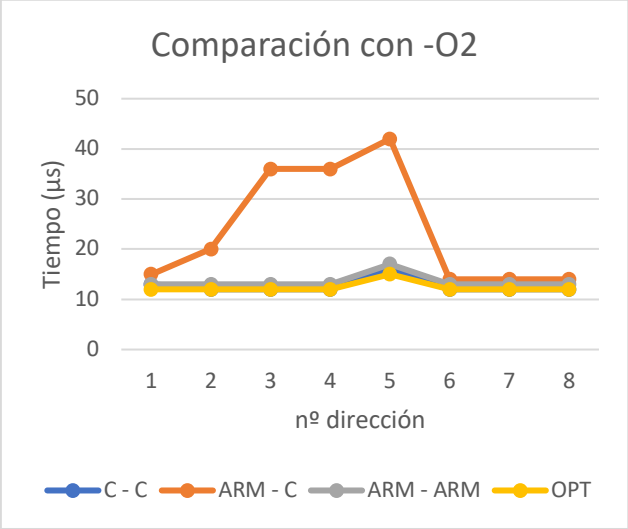

Figura 18

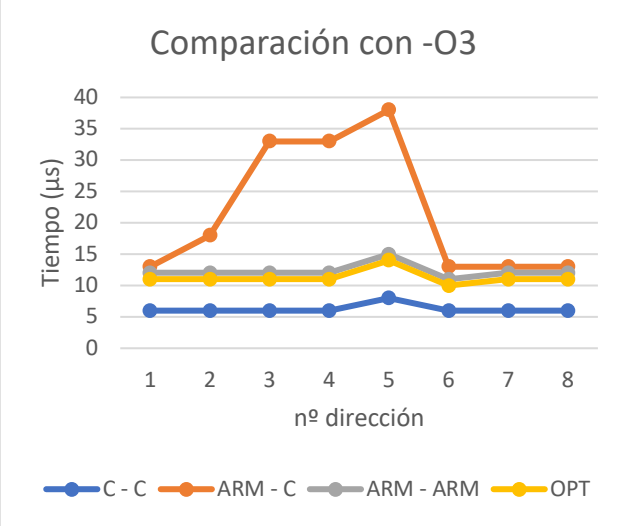

Figura 17

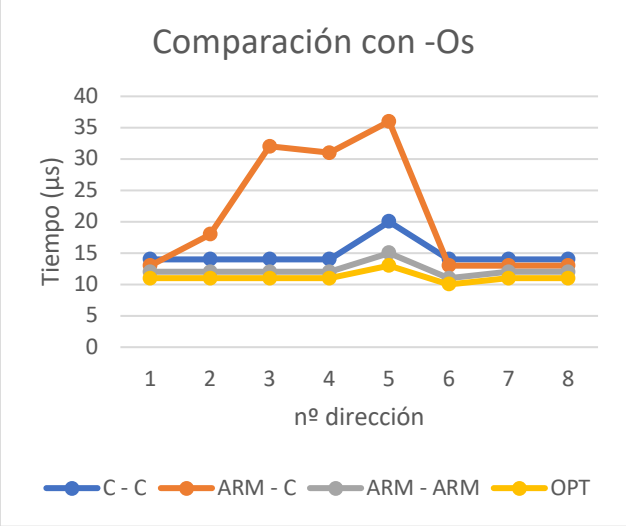

Figura 19

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.

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.

A continuación, se compararon los tamaños en bytes del código generado por el compilador:

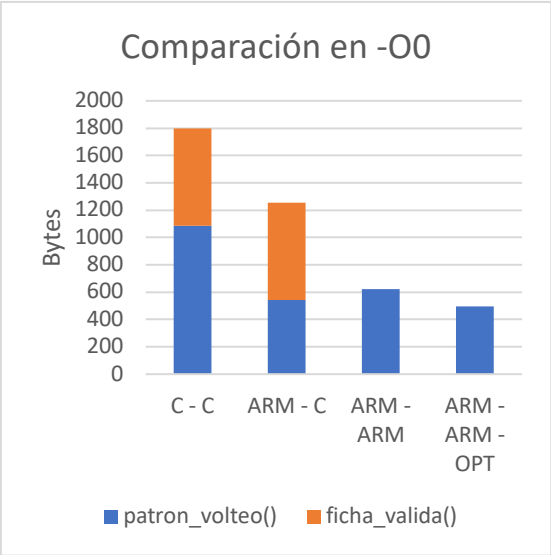

Figura 20

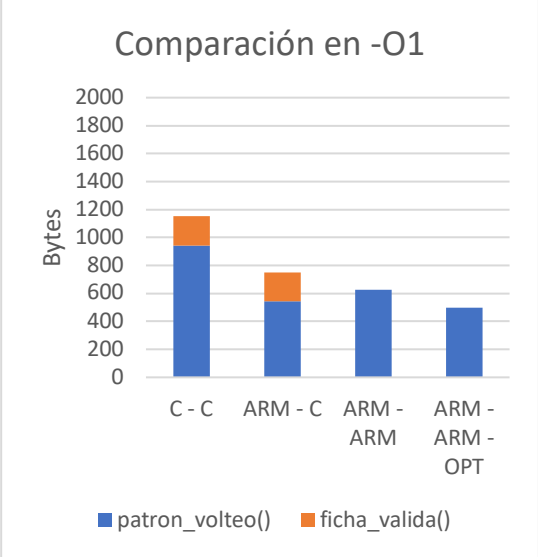

Figura 21

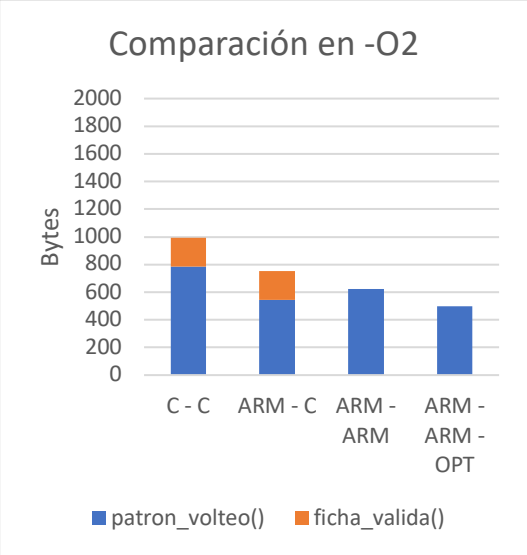

Figura 22

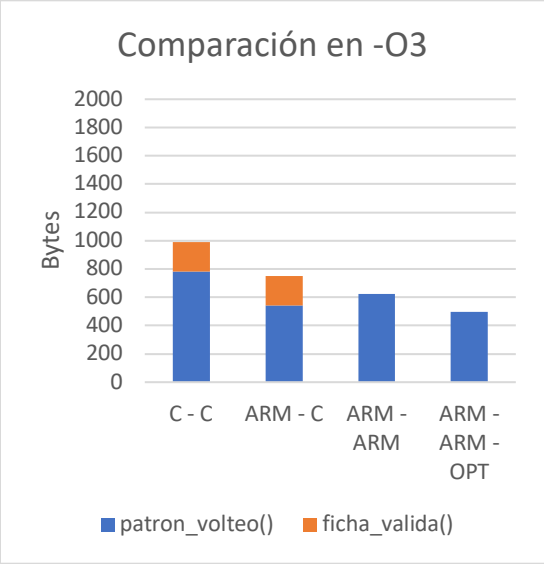

Figura 23

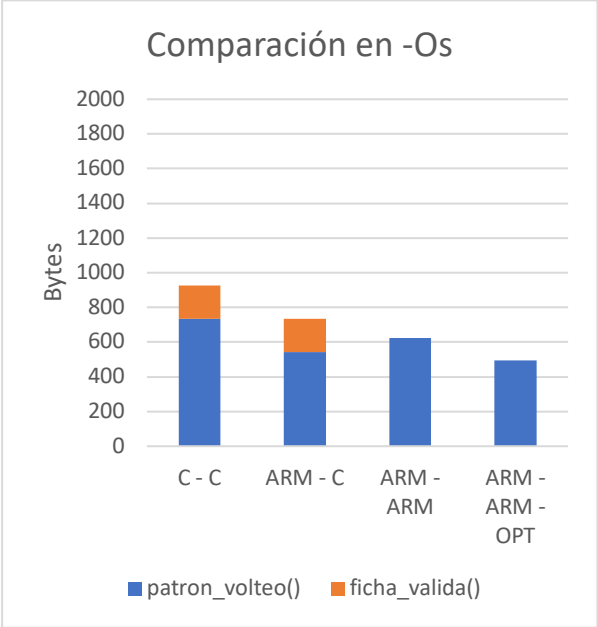

Figura 24

Lo primero que se ve es que sólo hay mejoras en cuanto al código en C, el ensamblador escrito a mano se mantiene exactamente igual, siempre ocupa lo mismo.

Analizando el código en ensamblador generado por el compilador, se ha visto que en -O1 no se realizan desenrollados de código. Sin embargo, en -O2 si se desenrolla el `while`, a pesar de esto el código generado con -O2 ocupa menos que el de -O1 porque se hace mejor uso de los registros y se evitan muchas operaciones de guardado y lectura

En Os no hay despliegue, pero se siguen aplicando las optimizaciones en cuanto a la optimización de registros, por lo que consigue un buen rendimiento y es la versión C-C que menos ocupa.

Para terminar las pruebas en cuanto a opciones de compilación, se probó a jugar al reversi usando las optimizaciones. Hizo falta declarar las variables `fila`, `columna` y `ready` como `volatile`, para que el compilador no las optimizara e impidiera cambiar sus valores de forma externa en tiempo de ejecución.

Apartado opcional 1

Una vez realizada la función ARM-ARM se vio que había varias comparaciones duplicadas, así que se eliminaron para ganar algo de eficiencia. Se juntaron las comparaciones correspondientes a `ficha_valida()` y `patron_volteo()` para que no hubiera que realizarlas dos veces. El código de la versión ARM-ARM optimizada se muestra a continuación:

```
@ Devuelve si ha encontrado un patrón de volteo y la longitud anali-
zada en el parámetro *longitud
patron_volteo_arm_arm_opt:
                        mov ip, sp
                        push {r4-r10, fp, ip, lr, pc}
                        sub fp, ip, #4
                        @ r0=@tablero
                        @ r1=@longitud
                        @ r2=FA
                        @ r3=CA
                        ldrsb r4,[fp,#4]  @r4 = SF
                        ldrsb r5,[fp,#8]  @r5 = SC
                        ldrb r6,[fp,#12]  @r6 = color

                        mov r9, #NO_HAY_PATRON  @ r9=resultado de la
                                                  subrutina
                        ldr r8, [r1] @ r8 = longitud inicial

                        @ FA = FA + SF
WHILE_ASM_OPT:          add r2, r2, r4
                        @ CA = CA + SC
                        add r3, r3, r5

                        add r10, r0, r2, LSL #DESPL @añadir fila
                        add r10, r10, r3  @añadir columna

@ ficha válida
@ FA < DIM && FA >= 0 && CA < DIM && CA >= 0 &&
(tablero[FA][CA] != CASILLA_VACIA)
                        cmp r2, #DIM
                        bge FIN_ASM_OPT   @ FA >= DIM
                        cmp r2, #0
                        blt FIN_ASM_OPT         @ FA < 0
                        cmp r3, #DIM
                        bge FIN_ASM_OPT         @ CA >= DIM
                        cmp r3, #0
                        blt FIN_ASM_OPT         @ CA < 0

                        ldrb r7, [r10]    @ r7=tablero[FA][CA]
                        cmp r7, #CASILLA_VACIA
                        beq FIN_ASM_OPT         @r7=CASILLA_VACIA


                        @ En este punto la posición es válida
                        cmp r7, r6
                        beq FIN_VALIDA_ASM_OPT
                                          @ tablero[FA][CA] == color
                        @ sumar 1 a longitud
                        add r8, r8, #1
                        b WHILE_ASM_OPT

FIN_VALIDA_ASM_OPT: cmp r8, #0
                            movne r9, #PATRON_ENCONTRADO

FIN_ASM_OPT:        str r8, [r1]@ Almacenar longitud calculada
                    mov r0, r9  @ Mover resultado de la subrutina a r0
                    ldmdb fp,{r4-r10, fp, sp, pc}
```

Código 14 - **patron_volteo_arm_arm_opt()**

Apartado opcional 2

Verificar a mano la versión en C para un único tablero y automatizar la verificación del resto de códigos no asegura la correcta ejecución siempre, para probar en profundidad los diseños realizados se desarrollaron una serie de pruebas.

Se hicieron dos conjuntos de pruebas. El primero comprueba que todas las implementaciones devuelven el mismo resultado ejecutando `patron_volteo_test()` sobre todas las casillas de diferentes tableros con fichas estratégicamente colocadas para probar la mayoría de posibilidades de encontrar y no encontrar patrón de volteo. El segundo conjunto de pruebas evalúa que el valor devuelto por las implementaciones es correcto. Primero se comprueba que las funciones se comporten correctamente cuando se introducen posiciones fuera del tablero. Después se plantean varios escenarios, tableros con diferentes fichas, se prueba a colocar una ficha en una esquina (posición 0,0), en un lado (posición 0,3) y en el centro (posición 3,3) y se comparan los resultado con el resultado real introducido a mano. Esta prueba se realiza tanto para fichas negras como blancas.

La ejecución de las pruebas fue satisfactoria, todas las implementaciones de `patron_volteo` pasaron los tests.

Las funcione correspondientes a las pruebas se pueden encontrar al final del fichero `reversi8_2019.c`, incluído entre los fuentes entregados.

5. Problemas encontrados y soluciones

Los principales problemas durante la práctica han sido la familiarización con el entorno de trabajo (placa y Eclipse ARM) y la gestión del tiempo de trabajo con la placa. El primer obstáculo se superó a base de trabajar con el entorno y de leer la documentación proporcionada e información en internet y preguntar a los profesores de la asignatura. En cuanto al segundo problema, se intentó ir a las sesiones prácticas con el trabajo más avanzado posible, de forma que pudiéramos dedicar el máximo de tiempo a hacer pruebas sobre la placa. Además, acudimos alguna vez al laboratorio fuera del horario asignado para terminar el trabajo.

6. Conclusiones

Esta primera parte del proyecto de la asignatura ha servido muy bien como herramienta para recordar algunos conceptos de la arquitectura de computadores, en concreto las asignaturas ya cursadas, o para introducir los entornos de desarrollo cruzados entre C y lenguaje ensamblador ARM.

No obstante, se han encontrado dificultades por el camino que gracias a la documentación se han sabido solucionar de forma solvente, aprendiendo siempre algo acerca de la arquitectura con la que se estaba trabajando y aplicándolo en el proyecto.

También ha servido para aprender a manejar bien los tiempos de un desarrollo sobre un hardware que no siempre está disponible, a redactar una memoria técnica y a optimizar.

En cuanto al código desarrollado, su comportamiento es el deseado y funciona de forma óptima sobre la placa de desarrollo, con unas medidas de coste en tiempo que permiten ver el trabajo de optimización que ha habido detrás, así como el calibrado correcto de `timer2` de acuerdo con las especificaciones del enunciado.

Para finalizar, resaltar que aunque la mayoría del código se haya desarrollado de forma satisfactoria, durante la sesión de corrección presencial se detectaron pequeños fallos, como que `timer2` debería medir dos veces el tiempo para evitar posibles subidas en `timer2_num_int` entre la lectura de interrupciones y la de ticks (Ver paso 7), o como la declaración innecesaria de los tableros de reversi como `static`, dos pequeños errores que ya han sido solucionados modificando ligeramente el código presentado en la entrega.

7. Referencias

[1] GuiaEntorno.pdf

[2] EntradaSalida.pdf

[3] P2-ec.pdf

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

[5] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html (consultado: 27/10/2019)

[6] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html (consultado: 27/10/2019)