Memoria Técnica Práctica 1

Proyecto Hardware

Jaime Yoldi Vigueras 779057 José Marín Díez 778148

Octubre 2020

Índice

1 Introducción **3**

2 Objetivos **3**

3 Tareas **3**
3.1 Mapa de Memoria 4
3.2 Implementación de funciones en ARM 5
3.2.1 patron_volteo_arm_c 5
3.2.2 patron_volteo_arm_arm 8
3.3 Verificación automática 10
3.4 Análisis de rendimiento y estudio del compilador 11

4 Conclusión **12**

5 Dificultades **13**

6 Anexo 1: Código **13**
6.1 patron_volteo_arm_c.s 13
6.2 patron_volteo_arm_arm.s 14
6.3 reversi8.c 16

1 Introducción

En este documento se presenta la memoria técnica correspondiente a la práctica 1 de la asignatura Proyecto Hardware. A lo largo del documento se detalla el proceso de optimización del conocido juego Reversi, partiendo de un proyecto proporcionado en lenguaje c. Para ello, algunas partes del código serán sustituidas por otras desarrolladas en *ARM*, para finalmente ser todo ello ejecutado con la ayuda de un emulador del procesador *ARM LPC2105*. El entorno de desarrollo empleado es *uVision IDE*.

También se realiza un análisis del código facilitado, analizando su coste y tamaño con ayuda del entorno *Keil* para después decidir cuáles son las partes más críticas y que por tanto deben ser optimizadas. Finalmente se comparan los resultados obtenidos con los del código generado por el propio compilador.

2 Objetivos

El principal objetivo de esta práctica es optimizar el rendimiento del famoso juego Reversi, cuyo código en c ha sido proporcionado, acelerando las funciones computacionalmente más costosas mediante el desarrollo de estas en lenguaje ensamblador.

Para lograr el objetivo, el trabajo se centra fundamentalmente en la función `patron_volteo` , la cual agrupa la mayor parte del tiempo y recursos durante la ejecución del programa. Para ello se requiere la comprensión de la función, así como el saber "comunicar" diferentes lenguajes para que trabajen conjuntamente, incluyendo paso de parámetros, uso de memoria, etc, siguiendo los estándares establecidos.

Es necesaria también la utilización de diferentes técnicas de optimización, como el *inlining*, además de dominar diversos aspectos del lenguaje c, como pueden ser el correcto uso de los tipos al más bajo nivel.

La práctica también busca aprender a utilizar el entorno, empleando de la manera correcta los recursos que proporciona, ya sea visualización de memoria o análisis de rendimiento, (en tiempo y espacio) y otros muchos más.

Finalmente, se necesita entender el funcionamiento de los sistemas empleados y cómo se estructura la memoria durante la ejecución del programa para distintos niveles de compilación y para cada técnica de de optimización empleada.

3 Tareas

Lo primero que se ha realizado es comprender el funcionamiento del código facilitado y ejecutarlo. Inicialmente el código no presentaba errores pero si *warnings*, estos en su mayoría se debían a los tipos empleados por algunas variables.

El más común se encuentra en la utilización de variables `char` , tipo que dependiendo de la máquina en la que es ejecutado el código utiliza una aritmética entera o natural. Para evitar el problema y que sea independiente de la máquina, se ha optado por especificar mejor los tipos en las variables más "problemáticas". Para ello, se ha hecho uso de los tipos definidos en `<stdint.h>` .

Algunas de las variables modificadas son `vSF` y `vSC` que contienen la tabla de direcciones y cuyos valores pueden ser tanto positivos como negativos, por lo que se han definido como `int8_t` . También se le ha asignado este tipo a las variables `fila` , `columna` y `ready` , las cuales normalmente son positivas, pero `fila` y `columna` de manera excepcional pueden tomar valor negativo, cuando se busca un patrón en una posición en el límite del tablero.

```
const int8_t vSF[DIM] = {-1,-1, 0, 1, 1, 1, 0,-1};
const int8_t vSC[DIM] = { 0, 1, 1, 1, 0,-1,-1,-1};
```

Además, y muy importante, es que a estas últimas se le ha añadido también la palabra clave `volatile` para obligar al programa a que lea esas variables siempre desde memoria ya que pueden ser modificadas en cualquier momento de manera externa a la ejecución del programa.

```
volatile static int8_t fila=0,
        columna=0,
        ready = 0;
```

Si esto no se cambia, en niveles altos de optimización el funcionamiento del programa ya no responde correctamente debido a que se lee el valor de la variable, cuyo puede haber sido modificado, en lugar de hacerlo directamente en memoria.

Por otra parte, hay algunas variables que siempre toman valor positivo, por lo que pueden ser declaradas como `unsigned` . Este es el caso de la matriz `tablero` cuyos valores sólo pueden ser: 0, 1 y 2. También todas las variables relacionadas con la fila y la columna cuando son pasadas como parámetros.

3.1 Mapa de Memoria

Al ejecutar el programa, en memoria se carga el código y las variables globales (declaradas fuera de las funciones), inicializadas y sin inicializar; en este orden. Además se reserva cierta parte de esta para guardar los datos de las funciones (*pila*), y otra parte para la utilización de memoria dinámica (*heap*).

La pila o *stack* y el *heap* se encuentran en el mismo fragmento de memoria, pero al guardar datos en la pila estos empiezan a almacenarse en las direcciones de memoria mayores hacia las direcciones menores, mientras que los datos que se guardan en el heap lo hacen al contrario, empiezan en las direcciones más pequeñas de este bloque. A continuación se muestra el esquema de memoria en orden ascendente de las direcciones de memoria.

Código Fuente
Variables inicializadas
Variables **no** inicializadas
Heap
...
Stack

La pila es donde se van guardando los datos, ya sea para mantener datos para el futuro de la ejecución, para evitar modificar registros que no se conoce si están siendo utilizados (estos se apilan y desapilar en el comienzo y final de las funciones) o para comunicar datos entre las distintas rutinas o funciones del programas En este caso, siguiendo el *ATPCS*, se guardan en la pila los argumentos que no se encuentren en los primeros cuatro registros. Además es donde se crean los bloques de activación para permitir las llamadas a funciones.

Este sería el esquema básico de un bloque:

Registros que no se pueden modificar
Variables locales
Frame pointer anterior
@ de retorno
Argumentos de la función

Sin embargo la implementación desarrollada de las funciones `patron_volteo` no exactamente este diseño, esto ha sido una decisión tomada para mejorar la eficiencia ya que el acceso a la pila es un acceso a memoria, las cuales han sido reducidas en la medida de lo posible (no es posible una eliminación completa).

Bloque de activación de `patron_volteo_arm_c` :

Registros utilizados
posicion_valida (variable local)
@ de retorno
SF
SC
color

Bloque de activación de `patron_volteo_arm_arm` :

Registros utilizados
SF
SC
color

3.2 Implementación de funciones en ARM

3.2.1 patron_volteo_arm_c

La función `patron_volteo_arm_c` es una función cuyo funcionamiento es idéntico a la función `patron_volteo` proporcionada en el fichero `reversi8.h` . Pero con la diferencia de que está escrita en ensamblador *ARM*, conservando, eso sí, la llamada a la función `ficha_valida` en lenguaje c.

Para implementar la función primero se ha de indicar al *linker* que esto es una función que va a ser usada desde otro fichero por lo que se le añade la directiva `EXPORT` , además le indicamos a la propia función que `ficha_valida` esta definida en otro fichero. También es necesario declarar la función en el archivo de código C con la directiva `extern` . Puesto que se trata de una combinación de dos lenguajes "enlazados" (ARM y C) tendremos que mantener el estandar del compilador, el *ATPCS* como ya hemos comentado.

```
PRESERVE8
    AREA codigo, CODE, READONLY
    EXPORT patron_volteo_arm_c
    IMPORT ficha_valida
```

Primero de todo, al igual que en todas funciones es necesario tratar el bloque de activación en el que se apila normalmente la dirección de retorno y el *frame pointer*.

En esta función se ha decidido omitir el uso del *frame pointer* por cuestiones de optimización. Su función, la de permitir acceder a los datos del bloque de activación va a ser realizada desde el *stack pointer* (sp). Esto se puede realizar debido a que durante el transcurso del programa solo se necesita leer los datos de la pila al comienzo, momento en el que todavía no se ha realizado ninguna otra llamada a función, lo que nos permite saber la posición de memoria en la que se encuentran los datos.

```
patron_volteo_arm_c
    PUSH {lr}
```

Como se observará más adelante, la función `patron_volteo` requiere el uso de una variable de tipo puntero la cual necesita un espacio en memoria. Para ello, la mejor manera y por la que se ha optado es la de reservarle un espacio de 4 bytes en la pila, en la posición anterior al campo que almacena la dirección de retorno.

```
    sub sp, sp, #4      ;reservar espacio para posicion_valida
```

Además, puesto que se trata de una función que contiene parámetros se necesita acceder a ellos. Estos van a ser guardados en registros sobreescribiendo lo que estos contienen, para ello antes

de guardar su valor se apila en contenido de los registros para no perderlo y recuperarlo cuando termine la subrutina.

```
PUSH {r4-r9}
```

Atendiendo al estándar *ATPCS* cuando se llama a una función escrita en ensamblador desde c, los primeros cuatro parámetros son escritos en los registros `r0-r3` y el resto son almacenados en el fondo del bloque de activación. Por ello, al comienzo de la función se tiene en `r0` la dirección en memoria del tablero; en `r1` el puntero a longitud y en `r2` y `r3` los valores de `FA` y `FC` respectivamente. Por otro, para obtener el resto de parámetros: `SF` , `SC` y `color` , se hace una lectura de los valores que se encuentran en la pila. Esto se hace, cómo se ha mencionado anteriormente en función de la cima de la pila, por lo que se fija un registro con el valor de donde se encuentran los parámetros para después leer los parámetros restantes. Esto se puede hacer mediante un `ldr` común, pero para optimizar tiempo se ha optado por una lectura múltiple en orden creciente de direcciones.

```
add r4,sp,#32   ;se fija el registro en el primer parámetro de la pila
ldm r4,{r4-r6}
```

Tras obtener los parámetros, se realiza una recolocación de `FA` , `CA` y `longitud` en los registros para cumplir con la llamada a la función `ficha_valida` y el estándar *ATPCS* en cuanto paso de parámetros a función.

```
;r1 = FA, r2 = CA, r7 = *longitud
mov r7,r1
mov r1,r2
mov r2,r3
```

Se guarda en `r3` el puntero `posicion_valida` al cual se le ha reservado un espacio en la pila y se preparan los valores de `FA` y `CA` (fila y columna de la casilla que se va a analizar) para pasarlos por registro a la función `ficha_valida` . Estos valores se guardan en la pila para posteriores llamadas a esta función.

```
add r3,sp,#24        ;r3 = @posicion valida
add r1,r1,r4         ;FA = FA + SF
add r2,r2,r5         ;CA = CA + SC

;casilla = r8 = ficha_valida(tablero, FA, CA, &posicion_valida)
PUSH {r0-r3}
bl ficha_valida
```

Tras el retorno de la función se procede a guardar el resultado que esta devuelve. Como estipula el *ATPCS* el valor retornado es devuelto en `r0` , por lo que se procede a guardar ese valor en en el registro encargado durante toda la subrutina de almacenar la casilla. Posteriormente, retoman los valores de los registros que `ficha_valida` ha modificado.

```
mov r8,r0
POP {r0-r3}
```

Se lee el valor correspondiente a la longitud, cuya dirección ha sido pasada por parámetro y guardada en el registro `r7` . Dado que inicialmente siempre toma el mismo valor, se opta por asignar al registro directamente el valor cero, evitando así una lectura innecesaria cuyo resultado es conocido.

```
mov r9, #0           ;r9 =longitud = 0
```

Ahora, se procede a buscar en todas las direcciones un patrón, esto se hace mientras la posición buscada sea válida y además contenga una ficha del color del rival. Para probar las condiciones en ensamblador se hacen encadenando las instrucciones `cmp` y `beq` , en la que se comprueba el valor de un parámetro y en caso de que no se cumpla se abandona el bucle. Para comprobar si la posición de una ficha es no válida, en lugar de leer el valor del puntero se hace una comparación

directamente con casilla, evitando así una lectura ya que se sabe que siempre que la posición no es válida, casilla toma valor 0.

```
while   cmp r8, #0      ;si la posición no es válida es porque
                        la casilla es vacía
        beq end_while
        cmp r8,r6       ;casilla != color
        beq end_while
```

Si ambas condiciones se cumplen se modifican las casillas de `ficha` y `columna` para hacer el mismo procedimiento en la siguiente casilla del supuesto patron. Además se incrementa en una unidad la `longitud` . El valor de la `longitud` es modificado en el registro que almacena su valor en lugar de hacerlo en su posición de memoria especificada por el puntero. Con esto se evita tener que hacer una escritura en cada iteración, ahorrando tiempo en un dato que solo necesita escribirse al terminar el bucle.

```
        add r1,r1,r4        ;FA = FA + SF
        add r2,r2,r5        ;CA = CA + SC
        add r9, r9, #1      ;logitud ++

        ;casilla = r8 = ficha_valida(tablero, FA, CA, &posicion_valida)
        PUSH {r0-r3}
        bl ficha_valida
        mov r8,r0
        POP {r0-r3}

        b while
```

Tras salir del `while` se comprueba si se ha encontrado un patrón o no, para ello se encadenan de nuevo `cmp` s y `beq` s para hacer las condiciones y devolver 1 si la última posición mirada era válida. Esto se da cuando la casilla es del color del jugador actual y además de longitud superior a 1. En caso contrario devuelve 0 ya que es no válida.

```
end_while
        mov r0, #0          ;r0 = NO_ENCONTRADO
        bne _else
        cmp r8,r6           ;casilla == color
        bne _else
        cmp r9, #0
```

Por razones de optimización en lugar de hacer un salto de una única línea se opta por realizar una operación condicional.

```
        movne r0, #1
```

En caso de que se haya encontrado un patrón, se guarda ya el valor de la longitud en su posición de memoria. Esto no sucede en caso de que el patrón no haya sido encontrado ya que el valor final de longitud es igual al que tomaba inicialmente (cero), por lo que sería ineficiente gastar tiempo y recursos en escribir lo mismo.

```
        str r9, [r7]        ;Se guarda la longitud
```

Finalmente, se concluye la subrutina, devolviendo a los registros `r4-r9` sus valores iniciales, liberando la memoria reservando para el puntero y asignando al `pc` la dirección de retorno, para salir de la función.

```
_else   POP {r4-r9}
        add sp, sp, #4
        POP{pc}
END
```

3.2.2 patron_volteo_arm_arm

El funcionamiento y objetivo de `patro_volteo_arm_arm` al igual que `patron_volteo_arm_c` es el mismo que el de `patron_volteo` escrito en lenguaje c. En este caso también está escrita en ensamblador *ARM* pero con la diferencia de que en lugar de incluir llamadas a `ficha_valida` , escrita en c, se opta por hacer uso de la técnica *inlining* que consiste en reemplazar la llamada a la función por el cuerpo de la propia función, el cual será escrito en ensamblador. Esta técnica supone una gran mejora en cuanto a optimización, siendo una técnica habitual empleada por los compiladores.

Para implementar la función, al igual que en la otra implementación y en todas las funciones, se ha de indicar al *linker* que es una función que va a ser usada desde otro fichero por lo que se le añade la directiva `EXPORT` . En este caso no se hace mención a que `ficha_valida` esta definida en otro fichero, por que no es así ya que se incluye dentro del propio código. De nuevo, al tratarse de una combinación de dos lenguajes, ARM y c se va a seguir el estándar *ATPCS*.

```
PRESERVE8
AREA cte, DATA
DIM EQU 8
AREA codigo, CODE, READONLY
EXPORT patron_volteo_arm_arm
```

Inicialmente, como siempre se trata el bloque de activación en el que se apila la dirección de retorno y el *frame pointer*.

En esta función también se ha considerado oportuno omitir el uso del *frame pointer*. De nuevo porque la función permite conocer la estructura completa del bloque de activación, pudiendo así, acceder a todos los datos del bloque desde la misma cima de la pila (sp). Además, como novedad, al tratarse de una función con *inlining* en la que no se hace llamadas a otras subrutinas, se puede evitar apilar la dirección de retorno debido a que el *link register*, que contiene el `pc` , no va a ser modificado durante toda la ejecución, por lo que su valor no va a cambiar y no requiere ser guardado en la pila.

```
;r0 = *tablero, r1 = *longitud, r2 = FA, r3 = CA,
;r4 = SF, r5 = SC, r6 = color

    patron_volteo_arm_arm
```

Puesto que los parámetros de la función tienen que ser guardados en registros, el valor que estos contienen va a ser almacenado en el bloque, para que no se pierda y recuperarlo cuando la subrutina concluya.

```
    PUSH {r4-r9}    ;guardo los registros que voy a utilizar
```

Como establece el *ATPCS*, los primeros cuatro parámetros son escritos en los registros `r0-r3` y el resto son almacenados en el fondo del bloque de activación. Para obtener estos últimos: `SF` , `SC` y `color` , se opta por hacer una lectura múltiple en función de `sp` , para ganar en eficiencia. Después "se preparan" los valores de `FA` y `CA` (fila y columna de la casilla que se va a analizar) para comenzar a buscar patrones.

```
    add r4, sp, #24 ;se fija el registro en el primer parámetro de la pila
    ldm r4,{r4-r6}  ;se hace una lectura múltiple de los 3 parámetros

    ;r8 = casilla, so se utiliza posición válida = (casilla > 0)
    add r2,r2,r4    ;FA = FA + SF
    add r3,r3,r5    ;CA = CA + SC
```

Como se puede apreciar, a diferencia del caso anterior donde no se hace *inline* de `ficha_valida` , no se requiere el uso de un puntero para `posicion_valida` ya que se puede realizar todo mediante condiciones.

Al implementar `ficha_valida` en *ARM* se ha tenido en cuenta, como en la función anterior, que la validez de la ficha depende del color de la casilla, por lo que este dato no se guarda si no que se calcula sobre el color de la casilla.

Para la implementación de `ficha_valida` se da por hecho que inicialmente la ficha no es válida, corrigiendo la decisión en caso de serlo. Para ello se inicializa a 0, valor que devuelve una ficha no válida. Después, se comprueban las condiciones, y si es válida se procede a leer qué casilla hay en esa posición.

En cuanto a la lectura de la posición, para mejorar la eficiencia del programa se hace uso de instrucciones condicionales, para evitar hacer un salto en caso de no válida. Además, al especificar la posición de memoria a leer se emplea uno de los recursos del procesador, el *barrel shifter* para el desplazamiento de bits, consiguiendo así multiplicar y sumar en una única instrucción de manera bastante eficiente.

```
        ;r8 = ficha_valida()
        mov r8, #0
        cmp r2, #DIM
        bhs noValida
        cmp r2, #0
        blt noValida
        cmp r3, #DIM
        bhs noValida
        cmp r3, #0
        addge r8, r3, r2, LSL #3
        ldrbge r8, [r0,r8]
```

De nuevo, sabiendo que `longitud` toma al principio un valor de cero, se le asigna directamente, evitando hacer una lectura.

```
        mov r9, #0      ;r9 =longitud = 0
```

Ahora, se procede a buscar en todas las direcciones un patrón, esto se hace mientras la posición buscada sea válida y además contenga una ficha del color del rival.

```
while   cmp r8, #0      ;pocicion_valida = 1
        beq end_while
        cmp r8,r6       ;casilla != color
        beq end_while
```

En caso de que ambas condiciones se cumplan se modifican los valores correspondientes para repetir el mismo procedimiento pero en la ficha siguiente. Además se incrementa en una unidad la longitud. Y al igual que en `patron_volteo_arm_c` esta es modificada en el registro y no en su posición de memoria evitando así una escritura por iteración.

```
        add r2,r2,r4    ;FA = FA + SF
        add r3,r3,r5    ;CA = CA + SC
        add r9, r9, #1  ; logitud ++

        ;r8 = ficha_valiada()
        mov r8, #0
        cmp r2, #DIM
        bhs end_while
        cmp r2, #0
        blt end_while
        cmp r3, #DIM
        bhs end_while
        cmp r3, #0
        blt end_while
        add r8, r3, r2 ,LSL #3
```

```
        ldrb r8, [r0,r8]

        b while
```

Tras salir del `while` se comprueba si se ha encontrado un patrón o no, para ello se comprueba el color y la longitud, devolviendo 1 si la casilla es del color del jugador actual y la longitud superior a uno (hay un un patrón) y 0 en caso contrario (no hay encontrado patrón).

```
end_while
        mov r0, #0      ;r0 = NO_ENCONTRADO
        cmp r8,r6       ;casilla == color
        bne _else
        cmp r9, #0
        movne r0, #1
```

En caso de que se haya encontrado un patrón, se guarda el valor de la longitud en su posición de memoria (haciendo uso de instrucciones condicionales). Si no se ha encontrado no se modifica ya que el valor final de longitud es igual al inicial (cero).

```
        strne r9, [r1]  ;Se guarda la longitud
```

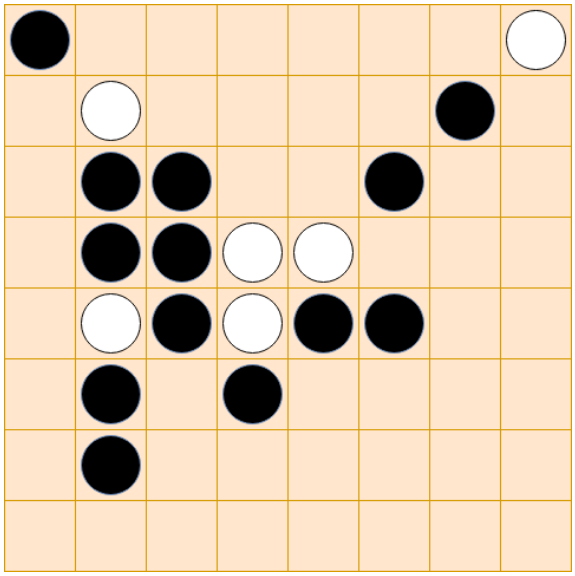
Finalmente, se concluye la subrutina, devolviendo a los registros `r4-r9` sus valores iniciales y asignando al pc la dirección de retorno guardada desde el principio en el *link register*, donde no se ha modificado como se ha mencionado previamente.

```
_else   POP {r4-r9}

        mov pc,lr
    END
```

3.3 Verificación automática

Para la verificación del código se decidió utilizar un tablero auxiliar en el que existían diversos patrones y sucesiones de fichas que no lo eran. Este se recorre buscando las fichas blancas y al encontrar una, comprueba que el resultado de las tres funciones, como la longitud devuelta, sean iguales. Este es el tablero con el que se trabajó en el test:



Este tablero tiene patrones de fichas blancas de distintos tamaños y direcciones para comprobar que la función no es susceptible a estos cambios. Junto al tablero nos encontramos con la función

`patron_volteo_test` con los mismos argumentos que `patron_volteo` que llama a las distintas versiones como ya se ha comentado.

```
int patron_volteo_test(uint8_t tablero[][DIM], int *longitud,
    uint8_t FA, uint8_t CA, int8_t SF, int8_t SC, char color){
```

Se declaran tres variables para guardar los resultados

```
int longitud1 = 0, longitud2 = 0, longitud3 = 0;
```

Se llama a las tres funciones

```
int i = patron_volteo_c_c(tablero, &longitud1, FA, CA, SF, SC, color);
int j = patron_volteo_arm_c(tablero, &longitud2, FA, CA, SF, SC, color);
int k = patron_volteo_arm_arm(tablero, &longitud3, FA, CA, SF, SC, color);
```

Si su ejecución no da el mismo resultado se introduce en un bucle infinito

```
    while(i!=j | j!=k){};
    if(i == 1)
        while(longitud1 != longitud2 | longitud2 != longitud3){}
    *longitud = longitud1;
    return i;
}
```

Como último aclarar que en la ejecución del test **no** se modifica el tablero, si se volteara habría patrones que desaparecerían o serían modificados antes de que el programa los identifica, esto no nos interesa porque el tablero está diseñado para que se ejecuten varios tipos de patrones que no queremos eliminar.

3.4 Análisis de rendimiento y estudio del compilador

Tanto para analizar las funciones más costosas, como para comprobar si las optimizaciones realizadas son útiles, ha sido necesario llevar a cabo un análisis de rendimiento.

Este análisis ha en realizar una comparativa del coste en tiempo y tamaño de código entre las distintas vesiones de las funciones `patron_volteo` y ficha_valida para la tarea de realizar el cálculo de ficha negra en la fila 2 columna 3.

Este análisis se hace con ayuda del `Performance Analyzer` que proporciona el emulador *Keil*.

Para ello se ha repetido el proceso estudiando el impacto de rendimiento y funcionamiento del código c en los distintos niveles de compilación y flags para las diferentes las funciones escritas en ensamblador *ARM*.

Primero con la opción `Optimize for Time` desactivada y luego activada, lo que ha requerido el uso del atributo `noinline` para forzar al compilador a que a niveles altos de compilación no haga *inlining* de `patron_volteo` y que la comparación sea lo más precisa posible.

```
int __attribute__ ((noinline)) patron_volteo_c_c(...) {
```

Por otra parte para obtener el tamaño en bytes de las mismas funciones se ha hecho uso de la herramienta `Code Coverage` también proporcionada por Keil.

Los resultados obtenidos se pueden ver en la siguiente tabla:

Función	Tiempo `patron_volteo` (us)	Tamaño `patron_volteo` (palabras [1])	Tiempo ficha_valida (us)	Tamaño ficha_valida (palabras [2])
O0	9.967	52	4.700	23
O0 -otime	9.967	52	4.700	23
O1	8.933	48	3.050	17
O1 -otime	8.933	48	3.050	17

Función	Tiempo `patron_volteo` (us)	Tamaño `patron_volteo` (palabras [1])	Tiempo ficha_valida (us)	Tamaño ficha_valida (palabras [2])
O2	8.017	36	3.100	16
O2 -otime	8.017	60	0.000	17
O3	8.017	36	3.100	16
O3 -otime	7.117	54	0.000	17
arm_c	8.817	38	___[3]	___[4]
arm_arm	6.900	43	0.000[5]	0.000[6]

Como se puede observar en la tabla, para los niveles de compilación O0 y O1, en los que no se lleva a cabo una optimización clara del código sino una "traducción" de las instrucciones C a ARM, el rendimiento es claramente inferior al resto de versiones.

Sin embargo, conforme el compilador aumenta el nivel de optimización del código (O2, O3), surgen códigos más rápidos, en ocasiones superando a la función arm_c tanto en tiempo como espacio.

El compilador con la opción Otime activada mejora mucho en tiempo, pero lo hace a cambio de un incremento en el tamaño.

Se ha podido observar que a niveles de optimización más altos (O2, O3) el compilador gcc hace uso de la técnica inline para ahorrar tiempo, técnica también usada en la función `patron_volteo_arm_arm` , donde se introduce la función `ficha_valida` en el mismo codigo de `patron_volteo` . Como resultado queda un código más largo pero más eficiente en tiempo.

Aun con todo el compilador no es capaz de mejorar en eficiencia a la función `patron_volteo_arm_arm` , implementada directamente en ensamblador.

4 Conclusión

Tras realizar `patron_volteo_arm_c` , `patron_volteo_arm_arm` y analizar los resultados obtenidos en cuanto a tiempo y uso de memoria se llega a la conclusión de que no es relativamente difícil mejorar la eficiencia en cuanto a tiempo de un programa generado por el compilador. Se pueden "rascar" microsegundos de muchas formas, incluso utilizando técnicas muy sencillas que no suponen ningún esfuerzo al programador, como puede ser simplemente utilizar instrucciones condicionales y operaciones de lectura múltiple, además de aprovechar bien todas los recursos del lenguaje como pueden ser los desplazamientos de bits.

Por otra parte, se ha podido observar que el hecho de reducir el bloque de activación de las subrutinas exclusivamente a lo necesario, evitando operaciones de apilar y desapilar, sorprendentemente supone un gran avance y realmente se consigue ahorrar mucho tiempo.

También es de destacar los grandes beneficios que en cuanto a tiempo supone el uso de inline en algunas funciones, ligado también a evitar la creación de subrutinas y lo que ello conlleva.

Sin embargo, en cuanto a uso de memoria hay que recalcar que es realmente difícil llegar a los niveles que el compilador ofrece en cuanto a este aspecto. Sobre todo cuando está configurado con los flags correspondientes para ello.

Finalmente, tras analizar todos los aspectos se puede llegar a la conclusión de que la función `patron_volteo_arm_arm` es mejor que la propuesta en c.

[1] Depende del procesador. En este caso palabras de 4 bytes.
[2] Depende del procesador. En este caso palabras de 4 bytes.
[3] Depende del nivel de optimización. Valores igual que `patron_volteo_arm_c` .
[4] Depende del nivel de optimización. Valores igual que `patron_volteo_arm_c` .
[5] Incluida en `patron_volteo_arm_arm` mediante inline.
[6] Incluida en `patron_volteo_arm_arm` mediante inline.

5 Dificultades

Durante la realización de esta práctica surgieron algunos problemas: el primero de ellos, tuvo que ver con el paso de parámetros. En la declaración de funciones de `reversi8.c` los argumentos eran pasados como chars, causando que a la hora de hacer lecturas el signo no se mantiene. Esto se soluciono cambiando el tipo `char` por `int8_t` (entero con signo de 8 bits).

Otro de los problemas tuvo que ver con el cálculo de posiciones de memoria. Al prescindir de la utilización del `fp` estos cálculos se complicaron y causaron fallos. Sin embargo, se corrigieron fácilmente al esquematizar la pila.

Aparte de estas pequeñas dificultades la práctica se ha desarrollado con fluidez.

6 Anexo 1: Código

6.1 patron_volteo_arm_c.s

La función `patron_volteo_arm_c` comprueba si hay que actualizar una determinada dirección, 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 que sirve para saber la longitud del patrón que se está analizando. Se usa para saber cuántas fichas habrá que voltear.

Al inicio de la función la dirección de tablero se encuentra en `r0` , la de longitud en `r1` , `FA` en `r2` y `CA` en `r3` . Mientras que el resto de parámetros se encuentran apilados . Tras desapilar y reorganizar `FA` y `CA` se encuentran en `r1` y `r2` respectivamente, @longitud en `r7` , `SF` y `SC` en `r4` y `r5` y color en `r6` . La función `ficha_valida` necesita la dirección de memoria para guardar un booleano la cual se encuentra en `r3` para no tener que moverla al llamar la función. `r8` guarda la variable casilla, que guarda el color que devuelve `ficha_valida` . `r9` guarda el valor que se debe guardar en longitud.

```
PRESERVE8
    AREA codigo, CODE, READONLY
    EXPORT patron_volteo_arm_c
    IMPORT ficha_valida

patron_volteo_arm_c
;r0 = *tablero, r1 = *longitud, r2 = FA, r3 = CA,
;r4 = SF, r5 = SC, r6 = color

        PUSH {lr}           ;se evita guardar fp
        sub sp, sp, #4      ;reservo espacio para posicion valida
        PUSH {r4-r9}        ;guardo los registros que voy a utilizar
        add r4,sp,#32       ;se fija el registro en el primer parametro de la pila
        ldm r4,{r4-r6}      ;se hace una lectura múltiple de los 3 parametros

        ;recolocación las variables
        ;r1 = FA, r2 = CA, r7 = *longitud
        mov r7,r1
        mov r1,r2
        mov r2,r3

        add r3,sp,#24       ;r3 = @posicion valida
        add r1,r1,r4        ;FA = FA + SF
        add r2,r2,r5        ;CA = CA + SC
```

```
        ;casilla = r8 = ficha_valida(tablero, FA, CA, &posicion_valida)
        PUSH {r0-r3}
        bl ficha_valida
        mov r8,r0
        POP {r0-r3}

        mov r9, #0          ; r9 = longitud = 0. Siempre 0 al comienzo.
while   cmp r8, #0          ; si no es valida la casilla siempre es vacia.
        beq end_while
        cmp r8,r6           ;casilla != color
        beq end_while
        add r1,r1,r4        ;FA = FA + SF
        add r2,r2,r5        ;CA = CA + SC
        add r9, r9, #1      ;longitud ++

        ;casilla = r8 = ficha_valida(tablero, FA, CA, &posicion_valida)
        PUSH {r0-r3}
        bl ficha_valida
        mov r8,r0
        POP {r0-r3}

        b while

end_while
        mov r0, #0          ;r0 = NO_ENCONTRADO
        bne _else
        cmp r8,r6           ;casilla == color
        bne _else
        cmp r9, #0
        movne r0, #1

        str r9, [r7]        ;Se guarda la longitud
_else   POP {r4-r9}
        add sp, sp, #4
        POP{pc}
    END
```

6.2 patron_volteo_arm_arm.s

La función `patron_volteo_arm_arm` comprueba si hay que actualizar una determinada dirección, 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 cuántas fichas habrá que voltear.

Al inicio de la función la dirección de tablero se encuentra en `r0` , la de `longitud` en `r1` , `FA` en `r2` y `CA` en `r3` . Mientras que el resto de parámetros se encuentran apilados . Tras desapilar `SF` y `SC` en `r4` y `r5` y `color` en `r6` . `r8` guarda el valor de la casilla que se está procesando y `r9` la longitud del patrón.

```
    PRESERVE8
    AREA cte, DATA
DIM     EQU     8
    AREA codigo, CODE, READONLY
```

```
    EXPORT patron_volteo_arm_arm

patron_volteo_arm_arm
;r0 = *tablero, r1 = *longitud, r2 = FA, r3 = CA,
;r4 = SF, r5 = SC, r6 = color


        PUSH {r4-r9}        ;registro para utilizar
        add r4, sp, #24     ;se fija el registro en el primer parametro de la pila
        ldm r4,{r4-r6}      ;se hace una lectura múltiple de los 3 parametros

;r8 = casilla, no se usa posicion valida = (casilla > 0)

        add r2,r2,r4        ;FA = FA + SF
        add r3,r3,r5        ;CA = CA + SC

        ;r8 = ficha_valiada()
        mov r8, #0
        cmp r2, #DIM
        bhs noValida
        cmp r2, #0
        blt noValida
        cmp r3, #DIM
        bhs noValida
        cmp r3, #0
        addge r8, r3, r2, LSL #3
        ldrbge r8, [r0,r8]

noValida
        mov r9, #0 ; r9 =longitud = 0


while   cmp r8, #0   ;pocicion_valida = 1
        beq end_while
        cmp r8,r6   ;casilla != color
        beq end_while
        add r2,r2,r4        ;FA = FA + SF
        add r3,r3,r5        ;CA = CA + SC
        add r9, r9, #1      ; logitud ++

        ;r8 = ficha_valiada()
        mov r8, #0
        cmp r2, #DIM
        bhs end_while
        cmp r2, #0
        blt end_while
        cmp r3, #DIM
        bhs end_while
        cmp r3, #0
        blt end_while
        add r8, r3, r2 ,LSL #3
        ldrb r8, [r0,r8]

        b while

end_while
```

```
        mov r0, #0  ;r0 = NO_ENCONTRADO
        cmp r8,r6   ;casilla == color
        bne _else
        cmp r9, #0
        movne r0, #1

        strne r9, [r1] ; Se guarda la longitud
_else   POP {r4-r9}
        mov pc,lr
    END
```

6.3 reversi8.c

```
// Tamaño del tablero
enum { DIM=8 };

// Valores que puede devolver la función patron_volteo())
enum {
    NO_HAY_PATRON = 0,
  PATRON_ENCONTRADO = 1
};

// Estados de las casillas del tablero
// deberia ser enum, lo dejamos como const para usar char...
//const char CASILLA_VACIA = 0;
//const char FICHA_BLANCA  = 1;
//const char FICHA_NEGRA   = 2;

enum {
    CASILLA_VACIA = 0,
    FICHA_BLANCA = 1,
    FICHA_NEGRA = 2
};

// candidatas: indica las posiciones a explorar
// Se usa para no explorar todo el tablero innecesariamente
// Sus posibles valores son NO, SI, CASILLA_OCUPADA
const char  NO              = 0;
const char  SI              = 1;
const char  CASILLA_OCUPADA = 2;

////////////////////////////////////////////////////////////////////////////////
// TABLAS AUXILIARES
// declaramos las siguientes tablas como globales para que sean más fáciles
//visualizarlas en el simulador __attribute__ ((aligned (8))): specifies a
//minimum alignment for the variable or structure field, measured in bytes,
//in this case 8 bytes

static const int8_t __attribute__ ((aligned (8))) tabla_valor[DIM][DIM] =
{
    {8,2,7,3,3,7,2,8},
    {2,1,4,4,4,4,1,2},
    {7,4,6,5,5,6,4,7},
    {3,4,5,0,0,5,4,3},
    {3,4,5,0,0,5,4,3},
    {7,4,6,5,5,6,4,7},
```

```
    {2,1,4,4,4,4,1,2},
    {8,2,7,3,3,7,2,8}
};


// Tabla de direcciones. Contiene los desplazamientos de las 8 direcciones posibles
const int8_t vSF[DIM] = {-1,-1, 0, 1, 1, 1, 0,-1};
const int8_t vSC[DIM] = { 0, 1, 1, 1, 0,-1,-1,-1};

////////////////////////////////////////////////////////////////////////////////////
// Variables globales que no deberían serlo
// tablero, fila, columna y ready son varibles que se deberían definir como
//locales dentro de reversi8. Sin embargo, las hemos definido como globales para que
//sea más fácil visualizar el tablero y las variables en la memoria
////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
// Tablero sin inicializar
////////////////////////////////////////////////////////////////////

static uint8_t __attribute__ ((aligned (8))) tablero[DIM][DIM] = {
    {CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,
    CASILLA_VACIA,CASILLA_VACIA},
    {CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,
    CASILLA_VACIA,CASILLA_VACIA},
    {CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,
    CASILLA_VACIA,CASILLA_VACIA},
    {CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,
    CASILLA_VACIA,CASILLA_VACIA},
    {CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,
    CASILLA_VACIA,CASILLA_VACIA},
    {CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,
    CASILLA_VACIA,CASILLA_VACIA},
    {CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,
    CASILLA_VACIA,CASILLA_VACIA},
    {CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,
    CASILLA_VACIA,CASILLA_VACIA}
};


////////////////////////////////////////////////////////////////////
// VARIABLES PARA INTERACCIONAR CON LA ENTRADA SALIDA
// Pregunta: ¿hay que hacer algo con ellas para que esto funcione bien?
// (por ejemplo añadir alguna palabra clave para garantizar que la sincronización a
//través de esa variable funcione)
volatile static int8_t fila = 0,
        columna = 0,
        ready = 0;


////////////////////////////////////////////////////////////////////////
// 0 indica CASILLA_VACIA, 1 indica FICHA_BLANCA y 2 indica FICHA_NEGRA
// pone el tablero a cero y luego coloca las fichas centrales.
void init_table(uint8_t tablero[][DIM], char candidatas[][DIM])
{
    int i, j;
```

```
    for (i = 0; i < DIM; i++)
    {
        for (j = 0; j < DIM; j++)
            tablero[i][j] = CASILLA_VACIA;
    }
#if 0
    for (i = 3; i < 5; ++i) {
    for(j = 3; j < 5; ++j) {
        tablero[i][j] = i == j ? FICHA_BLANCA : FICHA_NEGRA;
    }
    }

    for (i = 2; i < 6; ++i) {
    for (j = 2; j < 6; ++j) {
        if((i>=3) && (i < 5) && (j>=3) && (j<5)) {
        candidatas[i][j] = CASILLA_OCUPADA;
        } else {
        candidatas[i][j] = SI; //CASILLA_LIBRE;
        }
    }
    }
#endif
    // arriba hay versión alternativa
    tablero[3][3] = FICHA_BLANCA;
    tablero[4][4] = FICHA_BLANCA;
    tablero[3][4] = FICHA_NEGRA;
    tablero[4][3] = FICHA_NEGRA;

    candidatas[3][3] = CASILLA_OCUPADA;
    candidatas[4][4] = CASILLA_OCUPADA;
    candidatas[3][4] = CASILLA_OCUPADA;
    candidatas[4][3] = CASILLA_OCUPADA;

    // casillas a explorar:
    candidatas[2][2] = SI;
    candidatas[2][3] = SI;
    candidatas[2][4] = SI;
    candidatas[2][5] = SI;
    candidatas[3][2] = SI;
    candidatas[3][5] = SI;
    candidatas[4][2] = SI;
    candidatas[4][5] = SI;
    candidatas[5][2] = SI;
    candidatas[5][3] = SI;
    candidatas[5][4] = SI;
    candidatas[5][5] = SI;
}

////////////////////////////////////////////////////////////////////////////////
// Espera a que ready valga 1.
// CUIDADO: si el compilador coloca esta variable en un registro, no funcionará.
// Hay que definirla como "volatile" para forzar a que antes de cada uso la
//cargue de memoria

void esperar_mov(volatile int8_t *ready)
```

```
{
    while (*ready == 0) {};  // bucle de espera de respuestas hasta que el se modifique
                             // el valor de ready (hay que hacerlo manualmente)

    *ready = 0;  //una vez que pasemos el bucle volvemos a fijar ready a 0;
}

////////////////////////////////////////////////////////////////////////////////
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// IMPORTANTE: AL SUSTITUIR FICHA_VALIDA() Y PATRON_VOLTEO()
// POR RUTINAS EN ENSAMBLADOR HAY QUE RESPETAR LA MODULARIDAD.
// DEBEN SEGUIR SIENDO LLAMADAS A FUNCIONES Y DEBEN CUMPLIR CON EL ATPCS
// (VER TRANSPARENCIAS Y MATERIAL DE PRACTICAS):
//  - DEBEN PASAR LOS PARAMETROS POR LOS REGISTROS CORRESPONDIENTES
//  - GUARDAR EN PILA SOLO LOS REGISTROS QUE TOCAN
//  - CREAR UN MARCO DE PILA TAL Y COMO MUESTRAN LAS TRANSPARENCIAS
//    DE LA ASIGNATURA (CON EL PC, FP, LR,....)
//  - EN EL CASO DE LAS VARIABLES LOCALES, SOLO HAY QUE APILARLAS
//    SI NO SE PUEDEN COLOCAR EN UN REGISTRO.
//    SI SE COLOCAN EN UN REGISTRO NO HACE FALTA
//    NI GUARDARLAS EN PILA NI RESERVAR UN ESPACIO EN LA PILA PARA ELLAS
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
////////////////////////////////////////////////////////////////////////////////
// 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(uint8_t tablero[][DIM], int8_t f, int8_t 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;
}
// ejemplo de declaración de una función definida externamente:

////////////////////////////////////////////////////////////////////////////////
// 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
extern int patron_volteo_arm_arm(uint8_t tablero[][8], int *longitud, uint8_t f,
                                 uint8_t c, int8_t SF, int8_t SC, char color);
extern int patron_volteo_arm_c(uint8_t tablero[][8], int *longitud, uint8_t f,
                               uint8_t c, int8_t SF, int8_t SC, char color);
int __attribute__ ((noinline)) patron_volteo_c_c(uint8_t tablero[][DIM], int *longitud,
                                 uint8_t FA, uint8_t CA, int8_t SF, int8_t 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 test ejecuta las tres versiones de patron volteo, comprueba si
// son igules los resultados y devueve lo mismo que cuarquiera de las tres
//si algun resultado no coincide bloquea al programa(while infinito)
// 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_test(uint8_t tablero[][DIM], int *longitud, uint8_t FA, uint8_t CA,
                       int8_t SF, int8_t SC, char color) {

    int longitud1 = 0;int longitud2 = 0;int longitud3 = 0;

    int i = patron_volteo_c_c( tablero, &longitud1, FA, CA, SF, SC, color);
    int j = patron_volteo_arm_c( tablero, &longitud2, FA, CA, SF, SC, color);
    int k = patron_volteo_arm_arm( tablero, &longitud3, FA, CA, SF, SC, color);
```

```
    while(i!=j | j!=k){}
        if(i == 1)while(longitud1 != longitud2 | longitud2 != longitud3){}
        *longitud = longitud1;
    return i;
}
////////////////////////////////////////////////////////////////////////////////
// voltea n fichas en la dirección que toque
// 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
// FA y CA son la fila y columna a analizar
void voltear(uint8_t tablero[][DIM], char FA, char CA, char SF, char SC, int n, char color)
{
    int i;

    for (i = 0; i < n; i++)
    {
        FA = FA + SF;
        CA = CA + SC;
        tablero[FA][CA] = color;
    }
}
////////////////////////////////////////////////////////////////////////////////
// comprueba si hay que actualizar alguna ficha
// no comprueba que el movimiento realizado sea válido
// f y c son la fila y columna a analizar
// char vSF[DIM] = {-1,-1, 0, 1, 1, 1, 0,-1};
// char vSC[DIM] = { 0, 1, 1, 1, 0,-1,-1,-1};
int actualizar_tablero(uint8_t tablero[][DIM], char f, char c, char color)
{
    char SF, SC; // cantidades a sumar para movernos en la dirección que toque
    int i, flip, patron;

    for (i = 0; i < DIM; i++) // 0 es Norte, 1 NE, 2 E ...
    {
        SF = vSF[i];
        SC = vSC[i];
        // flip: numero de fichas a voltear
        flip = 0;
        patron = patron_volteo_c_c(tablero, &flip, f, c, SF, SC, color);
        //patron_volteo(tablero, &flip, f, c, SF, SC, color);
        //printf("Flip: %d \n", flip);
        if (patron == PATRON_ENCONTRADO )
        {
            voltear(tablero, f, c, SF, SC, flip, color);
        }
    }
    return 0;
}
//recorre el tablero <tablero> buscando fichas blancas,
//cuando encuentra una llama a patron volteo test
//tablero es un matriz 8*8 que representa un tablero de reversi8
void test (uint8_t tablero[][DIM]){
        /*int flip = 0;
        int patron = patron_volteo_test(tablero,&flip,1,1,1,0, FICHA_BLANCA);*/
        int i,j,k;
    for( i = 0; i < DIM; i++){
```

```
        for( j = 0; j < DIM; j++){
            if(tablero[i][j] == FICHA_BLANCA){
                for (k = 0; k < DIM; k++){
                    int8_t SF = vSF[k];
                    int8_t SC = vSC[k];
                    // flip: numero de fichas a voltear
                    int flip = 0;
                    int patron = patron_volteo_test(tablero, &flip, i, j, SF, SC, FICHA_BLANCA);
                }
            }
        }
    }

}


////////////////////////////////////////////////////////////////////////////////
// Recorre todo el tablero comprobando en cada posición si se puede mover
// En caso afirmativo, consulta la puntuación de la posición y si es la mejor
// que se ha encontrado la guarda
// Al acabar escribe el movimiento seleccionado en f y c

// Candidatas
// NO    0
// SI    1
// CASILLA_OCUPADA 2
int elegir_mov(char candidatas[][DIM], uint8_t tablero[][DIM], char *f, char *c)
{
    int i, j, k, found;
    int mf = -1; // almacena la fila del mejor movimiento encontrado
    int mc;      // almacena la columna del mejor movimiento encontrado
    char mejor = 0; // almacena el mejor valor encontrado
    int patron, longitud;
    char SF, SC; // cantidades a sumar para movernos en la dirección que toque

    // Recorremos todo el tablero comprobando dónde podemos mover
    // Comparamos la puntuación de los movimientos encontrados y nos quedamos con el mejor
    for (i=0; i<DIM; i++)
    {
        for (j=0; j<DIM; j++)
        {   // indica en qué casillas quizá se pueda mover
            if (candidatas[i][j] == SI)
            {
                if (tablero[i][j] == CASILLA_VACIA)
                {
                    found = 0;
                    k = 0;

                    // en este bucle comprobamos si es un movimiento válido
                    // (es decir si implica voltear en alguna dirección)
                    while ((found == 0) && (k < DIM))
                    {
                        SF = vSF[k];    // k representa la dirección que miramos
                        SC = vSC[k];    // 1 es norte, 2 NE, 3 E ...

                        // nos dice qué hay que voltear en cada dirección
```

```
                        longitud = 0;
                        patron = patron_volteo_test(tablero, &longitud,
                                                    i, j, SF, SC, FICHA_BLANCA);
                        //patron_volteo(tablero, &longitud, i, j, SF, SC, FICHA_BLANCA);
                        //  //printf("%d ", patron);
                        if (patron == PATRON_ENCONTRADO)
                        {
                            found = 1;
                            if (tabla_valor[i][j] > mejor)
                            {
                                mf = i;
                                mc = j;
                                mejor = tabla_valor[i][j];
                            }
                        }
                        k++;
                        // si no hemos encontrado nada probamos con la siguiente dirección
                    }
                }
            }
        }
    }
    *f = (char) mf;
    *c = (char) mc;
    // si no se ha encontrado una posición válida devuelve -1
    return mf;
}
////////////////////////////////////////////////////////////////////////////////
// Cuenta el número de fichas de cada color.
// Los guarda en la dirección b (blancas) y n (negras)
void contar(uint8_t tablero[][DIM], int *b, int *n)
{
    int i,j;

    *b = 0;
    *n = 0;

    // recorremos todo el tablero contando las fichas de cada color
    for (i=0; i<DIM; i++)
    {
        for (j=0; j<DIM; j++)
        {
            if (tablero[i][j] == FICHA_BLANCA)
            {
                (*b)++;
            }
            else if (tablero[i][j] == FICHA_NEGRA)
            {
                (*n)++;
            }
        }
    }
}

void actualizar_candidatas(char candidatas[][DIM], char f, char c)
{
```

```
    // donde ya se ha colocado no se puede volver a colocar
    // En las posiciones alrededor sí
    candidatas[f][c] = CASILLA_OCUPADA;
    if (f > 0)
    {
        if (candidatas[f-1][c] != CASILLA_OCUPADA)
            candidatas[f-1][c] = SI;

        if ((c > 0) && (candidatas[f-1][c-1] != CASILLA_OCUPADA))
            candidatas[f-1][c-1] = SI;

        if ((c < 7) && (candidatas[f-1][c+1] != CASILLA_OCUPADA))
            candidatas[f-1][c+1] = SI;
    }
    if (f < 7)
    {
        if (candidatas[f+1][c] != CASILLA_OCUPADA)
            candidatas[f+1][c] = SI;

        if ((c > 0) && (candidatas[f+1][c-1] != CASILLA_OCUPADA))
            candidatas[f+1][c-1] = SI;

        if ((c < 7) && (candidatas[f+1][c+1] != CASILLA_OCUPADA))
            candidatas[f+1][c+1] = SI;
    }
    if ((c > 0) && (candidatas[f][c-1] != CASILLA_OCUPADA))
        candidatas[f][c-1] = SI;

    if ((c < 7) && (candidatas[f][c+1] != CASILLA_OCUPADA))
        candidatas[f][c+1] = SI;
}



////////////////////////////////////////////////////////////////////////////
// Proceso principal del juego
// Utiliza el tablero,
// y las direcciones en las que indica el jugador la fila y la columna
// y la señal de ready que indica que se han actualizado fila y columna
// tablero, fila, columna y ready son variables globales aunque deberían ser locales de reversi8,
// la razón es que al meterlas en la pila no las pone juntas, y así jugar es más complicado.
// en esta versión el humano lleva negras y la máquina blancas
// no se comprueba que el humano mueva correctamente.
// Sólo que la máquina realice un movimiento correcto.
/*

//este tablero se utiliza para llamar a la funcion test
//coniene varios patrones que sirven para comprobar la fiabilidad de la funcion patron volteo
static uint8_t __attribute__ ((aligned (8))) tablero_test[DIM][DIM] = {
    {FICHA_NEGRA  ,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA, CASILLA_VACIA,
    CASILLA_VACIA,FICHA_BLANCA },
    {CASILLA_VACIA,FICHA_BLANCA ,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,
    FICHA_NEGRA  ,CASILLA_VACIA},
    {CASILLA_VACIA,FICHA_NEGRA  ,FICHA_NEGRA  ,CASILLA_VACIA,CASILLA_VACIA, FICHA_NEGRA,
    CASILLA_VACIA,CASILLA_VACIA},
```

```
    {CASILLA_VACIA,FICHA_NEGRA  ,FICHA_NEGRA  ,FICHA_BLANCA ,FICHA_BLANCA, CASILLA_VACIA,
    CASILLA_VACIA,CASILLA_VACIA},
    {CASILLA_VACIA,FICHA_BLANCA ,FICHA_NEGRA  ,FICHA_BLANCA ,FICHA_NEGRA, FICHA_NEGRA
    ,CASILLA_VACIA,CASILLA_VACIA},
    {CASILLA_VACIA,FICHA_NEGRA  ,CASILLA_VACIA,FICHA_NEGRA  ,CASILLA_VACIA, CASILLA_VACIA,
    CASILLA_VACIA,CASILLA_VACIA},
    {CASILLA_VACIA,FICHA_NEGRA  ,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA, CASILLA_VACIA,
    CASILLA_VACIA,CASILLA_VACIA},
    {CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA,CASILLA_VACIA, CASILLA_VACIA,
    CASILLA_VACIA,CASILLA_VACIA}
        };

void reversi8()
{
    test(tablero_test);
    test(tablero_test);
    return;
}*/
void reversi8()
{

////////////////////////////////////////////////////////////////////////
// Tablero candidatas: se usa para no explorar todas las posiciones del tablero
// sólo se exploran las que están alrededor de las fichas colocadas
////////////////////////////////////////////////////////////////////////
    char __attribute__ ((aligned (8))) candidatas[DIM][DIM] =
    {
        {NO,NO,NO,NO,NO,NO,NO,NO},
        {NO,NO,NO,NO,NO,NO,NO,NO},
        {NO,NO,NO,NO,NO,NO,NO,NO},
        {NO,NO,NO,NO,NO,NO,NO,NO},
        {NO,NO,NO,NO,NO,NO,NO,NO},
        {NO,NO,NO,NO,NO,NO,NO,NO},
        {NO,NO,NO,NO,NO,NO,NO,NO},
        {NO,NO,NO,NO,NO,NO,NO,NO}
    };


    int done;      // la máquina ha conseguido mover o no
    int move = 0; // el humano ha conseguido mover o no
    int blancas, negras; // número de fichas de cada color
    int fin = 0;  // fin vale 1 si el humano no ha podido mover
                  // (ha introducido un valor de movimiento con algún 8)
                  // y luego la máquina tampoco puede
    char f, c;    // fila y columna elegidas por la máquina para su movimiento

    init_table(tablero, candidatas);

    while (fin == 0)
    {
        move = 0;
        esperar_mov(&ready);
        // si la fila o columna son 8 asumimos que el jugador no puede mover
        if (((fila) != DIM) && ((columna) != DIM))
        {
            tablero[fila][columna] = FICHA_NEGRA;
```

```
            actualizar_tablero(tablero, fila, columna, FICHA_NEGRA);
            actualizar_candidatas(candidatas, fila, columna);
            move = 1;
        }

        // escribe el movimiento en las variables globales fila columna
        done = elegir_mov(candidatas, tablero, &f, &c);
        if (done == -1)
        {
            if (move == 0)
                fin = 1;
        }
        else
        {
            tablero[f][c] = FICHA_BLANCA;
            actualizar_tablero(tablero, f, c, FICHA_BLANCA);
            actualizar_candidatas(candidatas, f, c);
        }
    }
    contar(tablero, &blancas, &negras);
}
```