Memoria

Práctica 1: Optimización en ensamblador ARM.

Proyecto Hardware
Grado en Ingeniería Informática
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Alicia Yasmina Albero Escudero (721288)
Iván Escuín González (684146)

Octubre de 2018

RESUMEN EJECUTIVO

Durante esta primera práctica se ha desarrollado y configurado un entorno de ejecución para el juego Reversi 8, siendo posible su ejecución sobre la placa Embest S3CEV40, además de varias optimizaciones haciendo uso de código en ensamblador ARM.

Tras programar el dispositivo timer2 de la placa, con el objetivo de medir tiempos en futuras optimizaciones del código,aclimatarse al entorno de trabajo con el IDE Eclipse, la placa y haber estudiado el código proporcionado para el proyecto y sus peculiaridades, se comenzó a trabajar en la optimización del código, centrándonos en concreto en la función *patron_volteo*. Para ello se desarrollaron las funciones en ensamblador *patron_volteo_arm_c*, que mantiene las llamadas a funciones en C, y *patron_volteo_arm_arm,* que usa puramente código en ARM.

Para comprobar el correcto funcionamiento del juego con estas funciones, y de paso automatizar el proceso de medición de tiempo, se crearon las funciones *patron_volteo_test,* donde se realiza una jugada dictada con los parámetros de la función en las diferentes versiones de patron_volteo y se comparan sus tiempos y resultados, e *init_tests* que prepara el tablero para distintos casos de prueba.

Tras medir los tiempos notamos una mejora de un 12% sobre la función original en ambas variantes, cosa que dejó de ser cierta a medida que aplicamos las optimizaciones o1,o2,etc del compilador.

Adicionalmente se ha obtenido de esta práctica la experiencia, o un primer contacto al menos, con el hecho de tener que trabajar con dispositivos físicos que no siempre están a nuestra disposición, y cómo prepararse para trabajar de esta forma para futuras prácticas.

TABLA DE CONTENIDOS

1. INTRODUCCIÓN

Se quiere lanzar un sistema que juegue al Reversi contra una persona sobre una placa ARM. Para ello se ha proporcionado un código en C del juego. El objetivo de este proyecto es optimizar el tiempo de ejecución del juego sobre la placa. Para ello se trabajará sobre las funciones críticas (en cuanto a tiempo de ejecución) del código fuente *reversi8_2015.c*, implementándolas en ensamblador para poder compararlas con sus versiones en C.
Estas funciones críticas son *patron_volteo* y *ficha_valida*, que aunque no tiene un coste computacional alto, es usada por *patron_volteo* en su algoritmo de búsqueda.

Para trabajar con el código proporcionado y la placa usada para ejecutar dicho código (Embest S3CEV40 con microprocesador Samsung S3C44B0X basado en ARM) se usará el entorno de desarrollo Eclipse con sus plugins y add-on necesarios para hacer posible la ejecución. Tanto el IDE como el compilador, debuggers y controladores necesarios han sido proporcionados por los profesores de la asignatura.

En nuestro caso trabajaremos con nuestros equipos propios, ambos usando Windows 10 y se intentará gestionar el código mediante git y github.

2. OBJETIVOS

El objetivo principal es conseguir optimizar la función *patron_volteo.* Tal como se ha comentado, esta función se implementará en ensamblador para poder reducir el tiempo de ejecución del juego.

Para poder corroborar que las funciones implementadas tienen un coste en tiempo menor que las originales es necesario medir con precisión el tiempo de ejecución. Por ello se implementarán las funciones necesarias para poder trabajar con el *timer2* de la placa. Además de poder obtener las medidas de tiempo, la implementación de estas funciones ayudará a la familiarización con la placa.

3. METODOLOGÍA

3.1. Timer 2

Puesto que vamos a optimizar partes del código, necesitaremos alguna forma de medida para saber si estamos realizando un trabajo correcto y cuantificar dicha optimización. Para ello usaremos uno de los dispositivos de la placa, el timer2, para medir los tiempos de ejecución del código en microsegundos.

Con este fin se desarrollará la librería **timer2.h** con las funciones *timer2_inicializar, timer2_empezar, timer2_leer y timer2_parar*, implementadas en **timer2.c.**

Guiándonos del ejemplo dado en **timer.c** y de la información presente en el manual de la placa[1] sobre los dispositivos timer (pág 234) configuramos el timer2 en *timer2_inicializar*:

Primero configurando el controlador de interrupciones con los registros **rINTMOD**, **rINCON** para habilitar las interrupciones IRQ y **rINTMSK** para habilitar el tratamiento de interrupciones del timer2, siguiendo el ejemplo de timer0. Una vez habilitadas las interrupciones, se establece cuál será la función que actuará como subrutina de servicio en **pISR_TIMER2**, en nuestro caso la función *timer2_ISR*.

Esta función simplemente actualiza la variable compartida **timer2_num_int,** que almacenará el número de interrupciones producidas por el timer, y desactiva el bit correspondiente al timer2 en **rI_ISPC** para borrar la petición de interrupción que ya ha sido tratada.

Pasaremos ahora a configurar el comportamiento del timer2 a través de los registros **rTCFG1** y **rTCFG0**. En el manual se nos informa de que el timer2 y timer3 comparten preescalador, configurable mediante los bits [15:8] de **rTCFG1** y que el divisor se configura en los bits [11:8] de **rTCFG0**. Puesto que buscamos la mayor precisión posible en el timer, intentaremos que la frecuencia de reloj sea lo mayor posible, para ello estableceremos mediante máscaras: **rTCFG0** con bits [15:8] a 0 (dividiendo la señal entre valor de preescalado +1) y **rTCFG1** con bits [11:8] a 0 (seleccionando el divisor ½ en el multiplexor). De esta forma tendremos una señal de reloj efectiva de 32 MHz en el timer.

Antes de empezar el timer es necesario establecer los valores con los que se va a realizar la cuenta y la comparación para producir interrupciones. Puesto que queremos que se produzcan el menor número de interrupciones en el sistema para no sobrecargarlo estableceremos el registro de inicio de cuenta **rTCNTB2** a su valor máximo 0xFFFF o 65535 en decimal y el registro de comparación **rTCMPB2** a 0.

Una vez configurado, solamente nos faltará indicar en el registro de control de los timers **rTCON** que no queremos que se invierta la salida (0 en bit 14) y establecer actualización manual (1 en bit 13) para que los registros del timer se actualicen una vez pongamos ese bit a 0.

Ahora, tras llamar a la función *timer2_inicializar* podermos llamar *timer2_empezar*, que cambiará el registro de control **rTCON** para poner a 0 el bit 13 (actualizar de forma manual), indicar que queremos *auto-reload*(1 en bit 15), es decir, que se reinicie la cuenta una vez generada una interrupción, y lo pondremos en marcha (1 en bit 12). También reseteamos la variable contador **timer2_num_init** y el registro intermedio de cuenta **rTCNTO2** para evitar lecturas erróneas.

Para realizar una lectura de tiempo mediante la función *timer2_leer,* esta obtendrá el tiempo de cuentas completas, **timer2_num_init** * valor de cada cuenta(**rTCNTB2**) y lo dividirá entre la frecuencia (en este caso 32MHz), sumándole a este tiempo el que ha transcurrido en la cuenta actual, **rTCNTB2** - **rTCNTO2** dividido también entre la frecuencia. Dado que en cálculo usamos la frecuencia en MHz, obtendremos microsegundos. El cálculo en la función tiene esta

pinta: timer2_num_int * (rTCNTB2 * 0.03125) + ((rTCNTB2 - rTCNTO2) * 0.03125). Usaremos decimales en lugar de fracciones para obtener más precisión.

La función *timer2_parar* simplemente pondrá un 0 en el bit 12 de **rTCON** parando la cuenta de timer2 y hará una llamada a *timer2_leer* para devolver el tiempo.

3.2. Marcos de pila

En este apartado se tratan los marcos de pila indicados en el guión de la práctica, estos son, el marco de pila presente en las funciones en ensamblador de *patron_volteo_arm_c* y *patron_volteo_arm_arm* (que son identicos) y el marco de pila a la hora de acceder a la subrutina de servicio de interrupción programada en el timer0, *timer_ISR.*

3.2.1. Función patron_volteo_arm

Tanto para *patron_volteo_arm_c* como para *patron_volteo_arm_arm*, los marcos de pila de la función serán como los mostrados en la Fig 1.

En la pila (1) vemos el estado al entrar en la función, tanto en la primera llamada desde *actualizar_tablero* o *elegir_mov* o desde una invocación recursiva. La pila (2) muestra el estado antes de realizar la llamada a *ficha_valida*, en este caso no será necesario el paso de parámetros a través de la pila, pues la totalidad de parámetros necesarios estarán en r0-r3. Sin embargo *ficha_valida* requiere el puntero a la variable **posicion_valida**, por lo que será necesario reservar espacio en la pila antes de la llamada para pasar en r3 la dirección de memoria que representará el puntero.

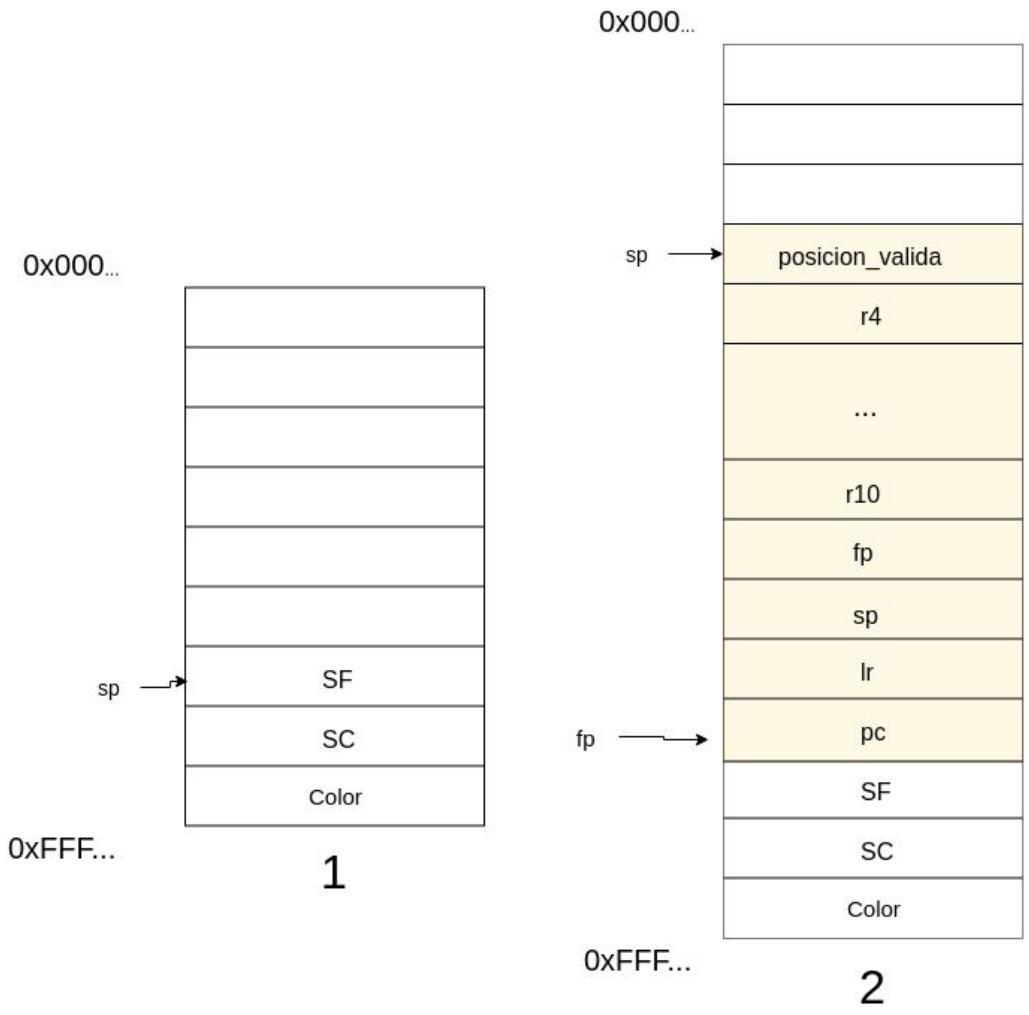

Fig 1. Marco de pila patron_volteo_arm

3.2.2. Función timer0 timer_ISR

Los movimientos de la pila en la subrutina de interrupción del timer0, descrita en el fichero **timer.c** son descritos en la Fig 2.

En la pila (3) se muestra el estado de la pila tras entrar en la subrutina y guardar el estado. Como en la subrutina es corta y sencilla no se realizan cambios en la pila, pero aún así el compilador recalcula la dirección de sp necesaria para recuperar el estado a través del fp (con la operación **sub sp,fp,#20**) en caso de que se hubiera usado la pila durante el tratamiento de la interrupción. En la pila (4) vemos los registros (estado) que se recuperan en rojo, y un movimiento adicional, marcado también en rojo, para recuperar el registro ip que almacena en el comienzo de la subrutina. Trás realizar el load múltiple cargará la dirección de ip en sp y simplemente realizará otro load adicional para recuperar ip y dejar sp en su estado inicial antes de producirse la interrupción.

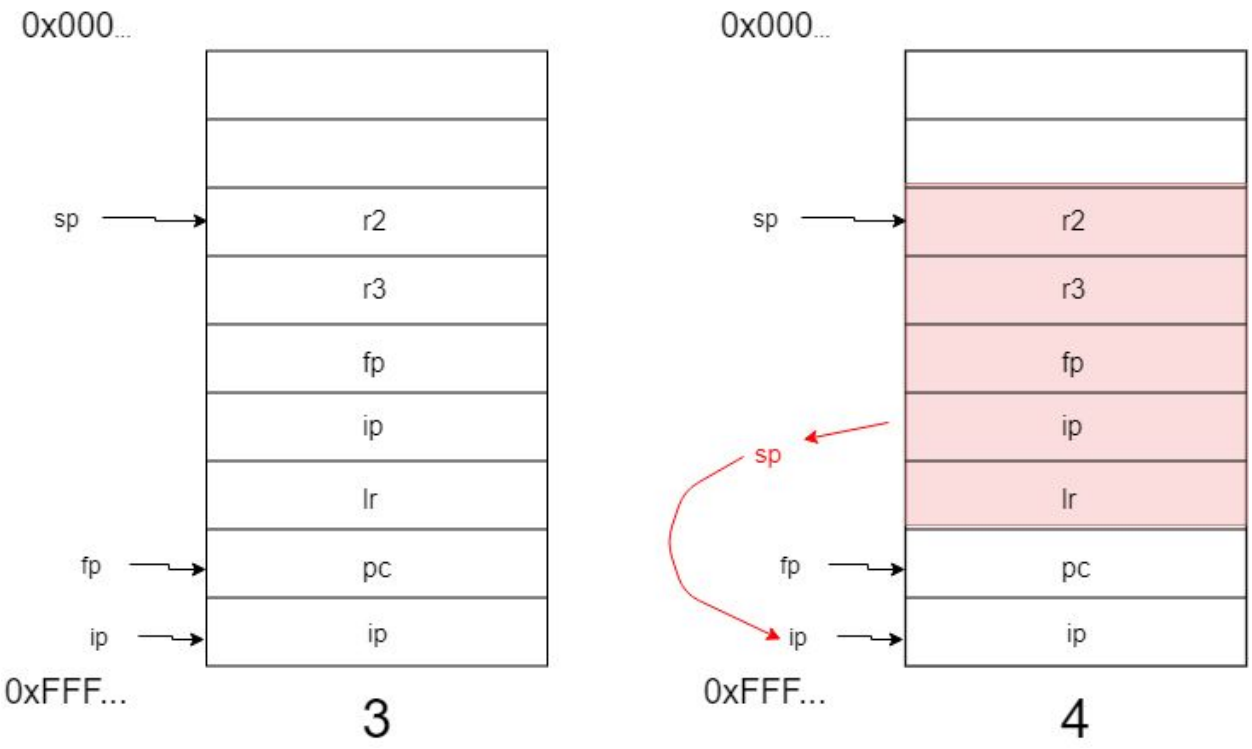

Fig 2. Marco de pila de timer_ISR

3.3. Código fuente

Se han realizado cambios en el fichero original de **reversi8_2018.c** con la finalidad de poder escoger el algoritmo de patron_volteo a la hora de jugar sin tener que cambiar cada llamada a la función a mano. Para ello se ha incluido una nueva constante **enum** que establece los valores **MODO_C, MODO_ARM_C y MODO_ARM_ARM**. Se declarará una variable **int modo_patron_volteo** en la función *reversi8* ,a la que se le asignará uno de los valores anteriores, y será pasada como parámetro a las funciones *actualizar_tablero* y *elegir_mov* que han sido modificadas, incluyendo un **switch** que en función del parámetro indicado usará una de las variantes de patron_volteo, dejando la función original en C como caso por defecto.

```
////////////////////////////////////////////////////////////////////////////
// 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(char tablero[][DIM], char f, char c, char color, int modo_patron_volteo)
{
    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;
        switch (modo_patron_volteo) {
        case MODO_ARM_C:
            patron = patron_volteo_arm_c(tablero, &flip, f, c, SF, SC, color);
            break;
        case MODO_ARM_ARM:
            patron = patron_volteo_arm_arm(tablero, &flip, f, c, SF, SC, color);
            break;
        default:
            patron = patron_volteo(tablero, &flip, f, c, SF, SC, color);
            break;
        }
        //printf("Flip: %d \n", flip);
        if (patron == PATRON_ENCONTRADO )
        {
            voltear(tablero, f, c, SF, SC, flip, color);
        }
    }
    return 0;
}
```

Se han creado dos funciones con la finalidad de realizar el testeo automático de las variantes de patrón volteo realizadas y tener una forma semi-automática (es necesario el uso de breakpoints y la lectura de variables en el debugger) de realizar las lecturas de tiempos mediante el timer2, las funciones *patron_volteo_test* e *init_test.*

La función *patron_volteo_test* recibirá los mismos parámetros que cualquiera de las variantes de patron_volteo y realizará una llamada a cada variante, guardando el resultado de cada una. Adicionalmente iniciará el timer2 (considerando que ya se ha inicializado en el main) y realizará dos lecturas para cada llamada, una antes y otra después, que guardará en el array **unsigned int tiempos [6]** para luego calcular el tiempo mediante una resta. Una vez obtenidos todos los resultados, realizará una comparación para comprobar que todas las funciones obtienen el mismo valor de retorno, resultando el bloqueo de la ejecución del programa mediante un bucle infinito en caso contrario.

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//  Funcion patron_volteo_test para comparar resultados entre las distintas implementaciones de patron_volteo.
//  Realiza mediciones de tiempo mediante el timer2 para cada caso de la funcion patron volteo y los almacena
//  en las variables time_c, time_arm_c y time_arm_arm para ser comprobados en debug. Estas variables deben ser
//  volatile para que puedan ser comprobadas en -o1 en adelante.
//
//  Si una de las funciones obtiene un resultado que no es igual a las demas la funcion se queda indefinidamente
//  en bucle.
// Parametros:
//      char tablero [][]: Array que contiene las fichas del juego
//      int* longitud: parametro por referencia que almacena la longitud del patron que se esta analizando
//      char FA: fila seleccionada
//      char CA: columna seleccionada
//      char SF: desplazamiento en la fila para buscar patron
//      char SC: desplazamiento en la columna para buscar patron
//      char color: tipo de ficha que ha iniciado la busqueda de patron
int patron_volteo_test(char tablero[][DIM], int *longitud, char FA, char CA, char SF, char SC, char color)
{

    unsigned int tiempos[6];    //Array para anotar tiempos del timer2
    volatile unsigned int time_c=0;
    volatile unsigned int time_arm_c=0;
    volatile unsigned int time_arm_arm=0;

    timer2_empezar();

    //Patron volteo c
    tiempos[0]=timer2_leer();
    int resultado_c=patron_volteo(tablero,longitud,FA,CA,SF,SC,color);
    tiempos[1]=timer2_leer();
    time_c=tiempos[1]-tiempos[0];

    //Patron volteo arm arm
    tiempos[4]=timer2_leer();
    int resultado_arm_arm=patron_volteo_arm_arm(tablero,longitud,FA,CA,SF,SC,color);
    tiempos[5]=timer2_leer();
    time_arm_arm=tiempos[5]-tiempos[4];


     //Patron volteo arm c
     tiempos[2]=timer2_leer();
     int resultado_arm_c=patron_volteo_arm_c(tablero,longitud,FA,CA,SF,SC,color);
     tiempos[3]=timer2_leer();
     time_arm_c=tiempos[3]-tiempos[2];

     timer2_parar();

     while(resultado_c != resultado_arm_c){}
     while(resultado_c != resultado_arm_arm){} //Breakpoint aqui
}
```

La función *init_test* se encargará de preparar los varios casos de test que se realizará mediante la función *patron_volteo_test*. Para ello recibirá mediante parámetros la dirección de tablero y de candidatas y modificará el tablero para cada caso de test. En cada caso se realizarán 6 llamadas a *patron_volteo_test*, una para cada posible dirección de búsqueda. Se tratan 4 posibles casos de test: El caso indicado en el enunciado de la práctica para realizar la medición de los tiempos, el tablero inicial con una ficha negra en [2,3], dos casos para comprobar que las funciones no fallan en los límites del tablero, uno con ficha negra en [0,0] y otro con ficha negra en [7,7] y un caso final para comprobar que las funciones son correctas en un caso no trivial de jugada, una ficha negra en [0,0], otra en [5,5] y en la diagonal que las une fichas blancas. Después de cada caso de test se reseteara el estado del tablero al estado inicial mediante la función *init_table.*

```
////////////////////////////////////////////////////////////////////////////////////////////////
//  Funcion que inicializa el tablero para realizar los tests necesarios y lo restaura a su estado
//  original antes de empezar a jugar.
//  Los casos de prueba realizados son:
//  · Tablero basico con ficha negra en 2,3. Usado para medir tiempos
//  · Tablero basico con ficha negra en 0,0.
//  · Tablero basico con ficha negra en 7,7.
//  · Tablero con diagonal, fichas negras en 0,0 y 5,5 y fichas blancas en (1,1),(2,2),(3,3) y (4,4)
//  Parametros:
//      char tablero[][]: Array que contiene las fichas del tablero de juego, al finalizar la funcion
//                        se devuelve a su estado original tras llamar a init_table
//      char candidatas[][]: Array que contiene el estado de las casillas del tablero de juego,
//                           al finalizar se devuelve a su estado original tras llamar a init_table
int init_test(char tablero[][DIM],char candidatas[][DIM]){

    //Caso de prueba basico, tablero inicial con ficha negra en 2,3
    tablero[2][3]=FICHA_NEGRA;
    candidatas[2][3]=CASILLA_OCUPADA;
    int longitud=0;
    patron_volteo_test(tablero,&longitud,2,3,-1,0,FICHA_NEGRA);
    patron_volteo_test(tablero,&longitud,2,3,1,0,FICHA_NEGRA); //Patron encontrado
    patron_volteo_test(tablero,&longitud,2,3,1,1,FICHA_NEGRA);  //Patron encontrado
    patron_volteo_test(tablero,&longitud,2,3,0,-1,FICHA_NEGRA);
    patron_volteo_test(tablero,&longitud,2,3,-1,-1,FICHA_NEGRA);
    patron_volteo_test(tablero,&longitud,2,3,0,1,FICHA_NEGRA);
    init_table(tablero,candidatas);

    //Caso de prueba, tablero inicial con ficha negra en 0,0. No se deberia encontrar patron.
    tablero[0][0]=FICHA_NEGRA;
    candidatas[0][0]=CASILLA_OCUPADA;
    longitud=0;
    patron_volteo_test(tablero,&longitud,0,0,-1,0,FICHA_NEGRA);
    patron_volteo_test(tablero,&longitud,0,0,1,0,FICHA_NEGRA);
    patron_volteo_test(tablero,&longitud,0,0,1,1,FICHA_NEGRA);
    patron_volteo_test(tablero,&longitud,0,0,0,-1,FICHA_NEGRA);
    patron_volteo_test(tablero,&longitud,0,0,-1,-1,FICHA_NEGRA);
    patron_volteo_test(tablero,&longitud,0,0,0,1,FICHA_NEGRA);
    init_table(tablero,candidatas);

    //Caso de prueba, tablero inicial con ficha negra en 7,7. No se deberia encontrar patron.
    tablero[7][7]=FICHA_NEGRA;
    candidatas[7][7]=CASILLA_OCUPADA;
    longitud=0;
    patron_volteo_test(tablero,&longitud,7,7,-1,0,FICHA_NEGRA);
    patron_volteo_test(tablero,&longitud,7,7,1,0,FICHA_NEGRA);
    patron_volteo_test(tablero,&longitud,7,7,1,1,FICHA_NEGRA);
    patron_volteo_test(tablero,&longitud,7,7,0,-1,FICHA_NEGRA);
    patron_volteo_test(tablero,&longitud,7,7,-1,-1,FICHA_NEGRA);
    patron_volteo_test(tablero,&longitud,7,7,0,1,FICHA_NEGRA);
    init_table(tablero,candidatas);

    //Caso de prueba, patron en diagonal con ficha negra en 0,0 y 5,5 y el resto fichas blancas
    tablero[0][0]=FICHA_NEGRA;
    candidatas[0][0]=CASILLA_OCUPADA;
    tablero[1][1]=FICHA_BLANCA;
    candidatas[1][1]=CASILLA_OCUPADA;
    tablero[2][2]=FICHA_BLANCA;
    candidatas[2][2]=CASILLA_OCUPADA;
    tablero[5][5]=FICHA_NEGRA;
    candidatas[5][5]=CASILLA_OCUPADA;
    longitud=0;
    patron_volteo_test(tablero,&longitud,0,0,1,0,FICHA_NEGRA);
    patron_volteo_test(tablero,&longitud,0,0,0,1,FICHA_NEGRA);
    patron_volteo_test(tablero,&longitud,0,0,1,1,FICHA_NEGRA); //Patron encontrado
    init_table(tablero,candidatas);

}
```

3.4. Optimizaciones

En el código de las funciones patron_volteo_arm_c() y patron_volteo_arm_arm() se ha optimizado el código reduciendo el número de operaciones load y store (ya que son las que mayor tiempo de ejecución tienen) así como el uso de load y store múltiples, reduciendo la complejidad del código. También se ha intentado utilizar el menor número de registros posible en todas las funciones en ensamblador.

Se ha hecho uso de instrucciones predicadas para determinar el valor de retorno de las variantes de patron_volteo mediante **movgt** y **movle** al escribir el valor en r0. Además, como el hilo de ejecución más frecuente es que no encuentre patrón, solo se salta condicionalmente a la invocación recursiva si se da el caso, precargando antes en r9 el valor de **longitud** tanto para realizar las comparaciones como para calcular **longitud + 1** antes de la invocación recursiva.

```
##########################################################################################################
#Funcion patron_volteo_arm_c
#Parametros: r0=@tablero, r1=@longitud, r2=FA, r3=CA, pila: SF,SC,color almacenados en ese orden
##########################################################################################################
.section .text
.global patron_volteo_arm_c
patron_volteo_arm_c:
    mov ip, sp
    stmdb sp!, {r4-r10,fp, sp, lr, pc}
    sub fp, ip, #4
    //Guardamos los parametros iniciales que vamos a modificar en otros registros
    mov r10,r0 //@tablero
    mov r9,r1 //@longitud
    //Inicializamos variables con las que vamos a trabajar
    //SF y SC se deben leer de la pila
    ldrsb r4,[fp,#4] //SF
    ldrsb r5,[fp,#8] //SC

    add r8,r4,r2 //FA = FA + SF
    add r7,r5,r3 //CA = CA + SC
    //Preparamos llamada a ficha valida, r0=tablero, r1=FA , r2=CA ,r3=@posicion_valida
    mov r0,r10
    mov r1,r8
    mov r2,r7
    //posicion_valida=0
    sub r3,sp,#4 //@posicion_valida
    mov r6,r3 //Nos guardamos la direccion para el retorno de la llamada

    bl ficha_valida
    //Volvemos de la funcion ficha_valida, casilla se encuentra en r0 y @posicion_valida en r4
    ldrb r3,[r6]
    cmp r3,#1 //posicion==1
    bne no_patron
    //cargamos color de la pila
    ldrb r6,[fp,#12] //r4=color
    cmp r0,r6 //casilla==color
    ldrb r0,[r9] //cargamos el valor de longitud
    bne llamada_recursiva
    cmp r0,#0 //longitud>0
    movgt r0,#1 //PATRON_ENCONTRADO
    movle r0,#0 //NO_HAY_PATRON
    b return_patron_volteo_arm_c

no_patron:
    mov r0,#0
    b return_patron_volteo_arm_c

llamada_recursiva:
    add r0,r0,#1    //*longitud=*longitud+1
    strb r0,[r9]

    mov r0,r10      //r0=@tablero
    mov r1,r9       //r1=@longitud
    mov r2,r8       //r2=FA
    mov r3,r7       //r3=CA
    stmdb sp!, {r4-r6}
    bl patron_volteo_arm_c

return_patron_volteo_arm_c:
    ldmdb fp,{r4-r10,fp,sp,pc}
    bx lr

##########################################################################################################
```

4. RESULTADOS

4.1. Problemas encontrados

Uno de los problemas encontrados fue la instalación de los drivers necesarios para la placa en Windows 10. Es necesario iniciar el equipo en modo avanzado con la opción de “Instalar controladores no firmados” activada. Esto se debe a que la documentación utiliza el sistema operativo Windows 7 mientras que para la realización del proyecto se ha utilizado Windows 10.

Otro de los problemas encontrados ha sido que el entorno no encontraba la variable del sistema PATH. Para solucionar este inconveniente ha sido necesario declarar la variable PATH dentro del entorno EclipseARM cada vez que se iniciaba.

Al realizar las mediciones de tiempo usando las optimizaciones del compilador -o1,-o2,etc nos dimos cuenta de que no éramos capaces de obtener los tiempos, ya que el compilador omitía las variables al no ser usadas en ninguna otra parte del código. Para solucionar este problema fué necesario declarar dichas variables como **volatile**.

4.2. Comparativa de tiempos y tamaño

Para realizar las diferentes comparativas y constatar la optimización del código en ensamblador se han realizado distintas medidas del tiempo de ejecución en seis escenarios. Todos parten de la misma posición de las fichas, la diferencia es el inicio de la función patrón volteo.En tablero para todos los escenarios se encuentran las 4 fichas centrales en la posición de inicio más una ficha negra en la posición 2,3. Los escenarios son los siguientes:

1. El patrón se inicia en -1,0 por lo que no encuentra patrón.
2. El patrón se inicia en 1,0, por lo que si encuentra patrón.
3. El patrón se inicia en 1,1 por lo que si encuentra patrón.
4. El patrón se inicia en 0,-1 por lo que no encuentra patrón.
5. El patrón se inicia en -1,-1 por lo que no encuentra patrón.
6. El patrón se inicia en 0,1 por lo que no encuentra patrón.

Con estos escenarios y optimización -o0 se obtienen las siguientes medidas de tiempo (en microsegundos).

escenario	1	2	3	4	5	6
patron_volteo_c	71	103	76	71	71	70
patron_volteo_arm_c	68	91	71	68	68	68
patron_volteo_arm	65	81	66	65	65	64

Tabla 1. Tiempos de ejecución de los distintos algoritmos (en microsegundos)

Se puede apreciar una reducción del tiempo en las funciones patron_volteo_arm_c y patron_volteo_arm_arm. Esto es debido a que la optimización del compilador de C en -o0 no optimiza algunas funciones de ensamblador como load y store múltiples. Tampoco tiene en cuenta el número de registros utilizados y apila todos los registros para las llamadas a las subrutinas.

Sin embargo, a medida que se comprueban distintas opciones de optimización se puede apreciar que la diferencia de tiempos de ejecución entre las funciones en C y ARM disminuye.

En esta tabla se recogen las medidas de tiempo de ejecución en microsegundos de los escenarios con optimización o1. La diferencia entre los tiempos en algunos escenarios en el caso de la función patron_volteo_arm_arm son incluso mayores que los de su homóloga en C.

escenario	1	2	3	4	5	6
timer2_c	57	71	59	59	58	57
timer2_arm_c	57	69	58	56	56	56
timer2_arm_arm	60	75	61	59	60	59

Tabla 2. Medidas de tiempo de ejecución con opción de compilación -o1

En posteriores optimizaciones (-o2,-o3,-os) las diferencia de tiempo entre ambas funciones son nulas y en varias ocasiones el tiempo de ejecución de la versión en ARM es mucho mayor que el de la versión en C, tal como puede apreciarse en las tablas 3,4 y 5.

escenario	1	2	3	4	5	6
timer2_c	53	58	55	53	53	53
timer2_arm_c	55	69	58	56	56	57
timer2_arm_arm	60	76	61	60	60	59

Tabla 3. Medidas de tiempo de ejecución con opción de compilación -o2

escenario	1	2	3	4	5	6
timer2_c	47	52	49	47	47	47
timer2_arm_c	56	69	57	55	55	56
timer2_arm_arm	59	74	60	59	60	59

Tabla 4. Medidas de tiempo de ejecución con opción de compilación -o3

escenario	1	2	3	4	5	6
timer2_c	54	65	54	53	53	53
timer2_arm_c	51	63	52	51	51	52
timer2_arm_arm	55	70	56	55	56	55

Tabla 5. Medidas de tiempo de ejecución con opción de compilación -os

Se puede observar que con la optimización -o3 (tabla 4), las funciones en ensamblador tardan más en ejecutar la función. Una de las razones por la que esta diferencia es tan acusada es la implementación de la función ficha_valida() por separado. En el caso de la función patron_volteo_arm_arm() no se ha incluido el código de ficha_valida() en su interior sino que se ha mantenido la llamada a la función. Por este motivo, se han generado más instrucciones load y store, que tienen un mayor tiempo de ejecución.

5. CONCLUSIONES

A través de esta práctica hemos adquirido un conocimiento inicial sobre un tema que hasta ahora había sido puramente teórico, el desarrollo de software para un hardware específico, con todas las peculiaridades que conlleva y sobre todo con la necesidad de planear cuidadosamente el tiempo, pues la disponibilidad de dicho hardware es limitada a la hora de realizar tests, y en particular las mediciones de tiempo en este proyecto.

Si bien hemos conseguido una mejora sobre la ejecución puramente en C, no hemos conseguido adaptar nuestras mejoras a las optimizaciones realizadas por el compilador, resultando en una desventaja con el código optimizado en C.

Aún así, las horas invertidas estudiando el código generado por el compilador, la documentación de la placa y cuadernos, el código fuente dado y el repaso de conocimientos previos nos ha dado una buena base donde asentarnos para enfrentarnos a las futuras prácticas, que nos acercan aún más a la placa.

BIBLIOGRAFÍA

- Aclaración de instrucciones load y store
    - https://azeria-labs.com/memory-instructions-load-and-store-part-4/
- Guía de usuario ensamblador
    - http://www.keil.com/support/man/docs/armclang_asm/armclang_asm_pge1427897406850.htm
- Información sobre las funciones en ensamblador
    - http://infocenter.arm.com/help/index.jsp
- Página de ayuda para la instalación de los drivers
    - https://winaero.com/blog/how-to-turn-off-and-disable-uac-in-windows-10/