

Conecta 4

Práctica 1: Desarrollo de código para el procesado ARM

Proyecto Hardware
Universidad de Zaragoza, curso 2022/2023
17/10/2022

	C1	C2	C3	C4	C5	C6	C7
F1							
F2							
F3							
F4							
F5							
F6							

Ayelen Nuño Gracia 799301
Loreto Matinero Augusto 796598



**Escuela de
Ingeniería y Arquitectura**
Universidad Zaragoza

ÍNDICE

0. Resumen:	3
1 Metodología	3
1.2 Análisis del funcionamiento	3
1.3 Implementación función en C	6
1.4 Mapa de memoria	6
1.5 Comparación marco de activación de la pila	7
1.6 Análisis del rendimiento	8
1.7 Programación de funciones en ARM	8
1.7.1 conecta4_buscar_alineamiento_arm	8
1.7.2 conecta4_hay_linea_arm_c	11
1.7.3 conecta4_hay_linea_arm_arm	13
1.8 Verificación automática y comparación de los resultados	15
2 Resultados	16
2.1 Medidas de rendimiento	16
2.2 Optimizaciones del compilador	17
3 Conclusiones	19

0. Resumen:

En este trabajo se van a realizar diversas implementaciones del juego conecta 4, buscando optimizar el rendimiento del juego acelerando las funciones computacionalmente más costosas. Para así familiarizarnos con el código en ARM, y aprender a analizar y medir los resultados obtenidos.

Las diversas versiones que se implementarán son:

- actualizar jugada C - hay cuatro? C - buscar línea C
- actualizar jugada C - hay cuatro? C - buscar línea ARM
- actualizar jugada C - hay cuatro? ARM - buscar línea C
- actualizar jugada C - buscar si hay cuatro ARM

Todo ello se realizará ateniéndonos al estándar ATPCS.

Para ello se ha utilizado el simulador keil, en concreto se trabajará sobre el procesador LPC2150. En el cual se ha estudiado el material proporcionado y se han afianzado los conocimientos sobre el encapsulamiento del código y el funcionamiento de la pila.

Tras lo cual se han implementado las versiones anteriormente mencionadas, por último se ha estudiado el coste temporal de la ejecución de cada una de las versiones, al igual que el tiempo de las funciones en c con diversos niveles de optimización.

De todo ello se ha extraído la conclusión de que el código en ensamblador directo tiende a ser más óptimo que el que se ha realizado a alto nivel siempre que el compilador no aplique ningún nivel de optimización posteriormente al código en alto nivel.

De igual forma se ha concluido que los diversos niveles de optimización eliminan y reorganizan el código llegando a obtener resultados sorprendentes.

Por último se han realizado pruebas automáticas de verificación dado que los errores en lenguaje ensamblador a veces son difíciles de detectar, y para que las mediciones realizadas, y el análisis posterior sean válidos que el código realizado sea funcional.

1 Metodología

Para el comienzo de este proyecto lo primero que se ha realizado es un estudio profundo del código proporcionado, además de la documentación del entorno con el que se va a trabajar. En este proyecto se van a seguir las pautas establecidas por ATPCS.

1.2 Análisis del funcionamiento

Las siguientes funciones *conecta4_jugar*, *conecta4_hay_linea_c_c* y *conecta4_buscar_alineamiento_c* son las que se encargan de gestionar el correcto funcionamiento del juego, es por ello que comprenderlas es fundamental para el desarrollo del proyecto.

conecta4_jugar:

Es la función llamada desde el main y consiste en un bucle while en el cual se van a leer de memoria las variables *entrada* y *columna* (variables introducidas directamente en memoria por el usuario). Una vez introducidas se comprueba si es posible colocar esa ficha,

comprobando tanto que la columna como la fila son válidas y se encuentran dentro de los límites establecidos.

Una vez introducida la ficha se actualiza el tablero, y se comprueba si se ha ganado la partida con un 4 en línea, mediante la función *C4_verificar_4_en_linea* que llama a la función *conecta4_hay_linea_c_c* y se queda en un bucle infinito si es así, quedando el programa bloqueado y dando por finalizada la partida. Si no se ha ganado la partida se comprueba la siguiente posibilidad y es que los dos jugadores hayan quedado en empate mediante la función *C4_comprobar_empate* que comprobará que la fila 6 está llena y no se pueden introducir más filas quedando así los dos jugadores en empate.

Finalmente si no se ha dado lugar a ninguno de estos casos, se cambia el color de la ficha para que la siguiente jugada sea realizada por un jugador distinto alternando así el turno entre jugadores.

conecta4_hay_linea_c_c:

Esta función es invocada cada vez que se realiza un movimiento para ver si se ha realizado 4 en línea y se ha ganado la partida. Para ello se le pasan como parámetros un tablero, y los datos de la última jugada realizada, es decir en qué fila y casilla se ha colocado la última ficha y de qué color es la misma.

A partir de esta información se va a recorrer el tablero desde la posición de la última casilla introducida (la que se nos ha pasado como parámetro) en todas las direcciones posibles. El orden en el que se realizan las comprobaciones son: horizontal, vertical y por último se comprueban las diagonales. Para ello hace uso de la función *conecta4_buscar_alineamiento_c*, pasándole la dirección que queremos comprobar.

Esta función nos devolverá el número de fichas consecutivas que hay en el tablero en una de las direcciones (por ejemplo fichas blancas hacia la derecha desde el punto en el que nos encontramos) en caso de que este número sea mayor o igual a 4, se devolverá true para indicar que el jugador ha ganado. En caso contrario se volverá a llamar a la función para saber el número de fichas que este jugador tiene en el mismo eje en el sentido inverso (en el ejemplo sería fichas blancas hacia la izquierda desde la posición de la ficha que nos pasan por parámetro) y el número de casillas se sumaría resultado de la primera invocación de *conecta4_buscar_alineamiento_c* para saber si se ha ganado el juego. En caso contrario se devolverá false a la función invocadora.

conecta4_buscar_alineamiento_c:

Como se ha explicado anteriormente esta función recursiva se encarga de devolver el número de fichas consecutivas que hay de un mismo color dada la posición exacta de la ficha que se acaba de introducir y la dirección en la cual se va a comprobar si hay más fichas del mismo color. Esta dirección viene dada por las variables *delta_fila* y *delta_columna* y mientras se encuentra una ficha del mismo color y que sigue esta dirección se aumenta el valor de estas variables en la dirección proporcionada hasta encontrar una celda que esté vacía, sea de distinto color o se llegue al borde del tablero.

Ejemplo de ejecución:

Partimos de la situación en la que la nueva jugada es la ficha blanca situada en la posición C5-F2. El programa principal llamará a *conecta4_jugar*, que detectará que hay una nueva jugada y tras recoger la información de esta, lo primero que hará será comprobar que la dirección introducida se encuentra dentro del tablero, a través de las funciones *C4_fila_valida(row)* y *C4_columna_valida(column)* respectivamente.
















Una vez realizada la comprobación se actualizará el tablero *C4_actualizar_tablero(cuadrícula_1,row,column,colour)* quedando el tablero que se ilustra en la siguiente imagen.

El último paso será comprobar si se ha ganado la partida con esta jugada. Para ello llamaremos a *4_verificar_4_en_linea(cuadrícula_1, row, column, colour)* quien en un bucle que se ejecutará 4 veces invocará a la función *conecta4_buscar_alineamiento_c*, que comprobará de la siguiente manera si hay línea.

Primero se comprobará la posición C4-F2, como la función *celda_color* devolverá negro en vez de blanco se devolvera el valor 0 a la función invocadora dado que no existe una consecución de fichas blancas, tras esto se volvera a invocar a la función *conecta4_buscar_alineamiento_c* pero comprobando la casilla C6-F2 en el que nos encontramos con una ficha negra por lo tanto se devolverá 0, en esta caso la longitud de las fichas consecutivas blancas es 0 por ello se pasará a comprobar las celdas verticales, y posteriormente las dos diagonales siguiendo el mismo procedimiento.

A continuación se especifica el caso de la comprobación vertical ya que plantea un nuevo escenario:

En el caso de las celdas verticales se comprobará en primer lugar que la ficha superior es una ficha blanca, en este caso sí que encuentra una ficha que coincide con su color, por ello se volverá a llamar recursivamente a la función *conecta4_buscar_alineamiento_c*, en esta segunda iteración se volverá a la función *conecta4_hay_linea_c_c* la cual cambiará las variables *delta_filas* y *delta_columnas* para que se realice la comprobación en el sentido inverso.

	C1	C2	C3	C4	C5	C6	C7
F1							
F2							
F3							
F4							
F5							
F6							

1.3 Implementación función en C

Como se requiere en el enunciado se ha realizado la implementación de la función *C4_comprobar_empate*. El código de la función es el siguiente:

```
int C4_comprobar_empate(CELDA cuadrícula[TAM_FILAS][TAM_COLS]) {
    for (int i = 0; i <= NUM_COLUMNAS; i++){
        if(celda_vacia(cuadrícula[NUM_FILAS][i])) {
            return(0);    // No hay empate
        }
    }
    return(1);    // Hay empate
}
```

Para comprobar si los dos jugadores han terminado en empate lo único que se necesita es verificar que la última fila de la cuadrícula está llena, por lo que ningún jugador puede añadir más fichas. No es necesario verificar dentro de esta función si se ha ganado la partida ya que esto se comprueba antes. Por ello la función es un simple bucle for que comprueba que ninguna celda de la fila 6 está vacía, si lo está estará dentro del if por lo que no hay empate y sino terminará el bucle devolviendo un 1, indicando así que los dos jugadores han quedado en empate.

1.4 Mapa de memoria

El programa va a comenzar en la dirección 0x0000063C, esta dirección la cual podemos observar en el registro pc al iniciar la simulación del programa, es donde encontraremos las instrucciones codificadas en memoria. A continuación se muestran los registros al inicio del programa: estos registros siguen el estándar ATPCS, por ello la funcionalidad de estos registros a lo largo de la ejecución del programa se ve reflejada en la siguiente imagen.

R0	0x400000A0	Register	Synonym	Special	Role in the procedure call standard
R1	0x400000A0	r15		PC	The Program Counter.
R2	0x400000A0	r14		LR	The Link Register.
R3	0x400000A0	r13		SP	The Stack Pointer.
R4	0x00000000	r12		IP	The Intra-Procedure-call scratch register.
R5	0x40000040	r11	v8	FP	ARM-state variable-register 8. ARM-state frame pointer.
R6	0x00000000	r10	v7	SL	ARM-state variable-register 7. Stack Limit pointer in stack-checked variants.
R7	0x00000000	r9	v6	SB	ARM-state v-register 6. Static Base in PID/re-entrant/shared-library variants
R8	0x00000000	r8	v5		ARM-state variable-register 5.
R9	0x00000000	r7	v4	WR	Variable register (v-register) 4. Thumb-state Work Register.
R10	0x00000668	r6	v3		Variable register (v-register) 3.
R11	0x00000000	r5	v2		Variable register (v-register) 2.
R12	0x0000063C	r4	v1		Variable register (v-register) 1.
R13 (SP)	0x400004A0	r3	a4		Argument/result/scratch register 4.
R14 (LR)	0x000000E8	r2	a3		Argument/result/ scratch register 3.
R15 (PC)	0x0000063C	r1	a2		Argument/result/ scratch register 2.
CPSR	0x600000D3	r0	a1		Argument/result/ scratch register 1.
SPSR	0x00000000				

En este caso encontramos especial interés en los registros que nos ayudarán a gestionar las subrutinas para encapsular el código, gestionar la pila, y compartir variables entre los distintos procesos.

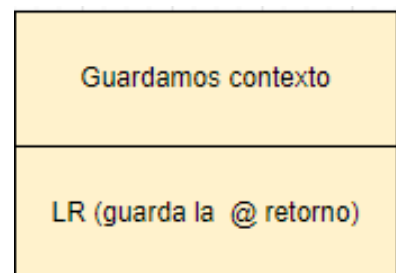
Otro aspecto fundamental de nuestro programa son las variables globales que se compartirán a lo largo del programa, en este caso encontramos dos tipos de variables globales fundamentales. la primera de ellas es la celda que representa. Cada celda se codifica en 8 bits. Estos 8 bits representan 3 posibles estados.

- bit 0: 1: ficha blanca; 0 : no hay ficha blanca
- bit 1: 1: ficha negra; 0 : no hay ficha negra
- bit 2: 0: posición vacía; 1: ocupada

El segundo atributo global sería el tablero, el cual está formado por una cuadrícula con las siguientes dimensiones 7x8, las cuales definirán el tablero en el que se va a jugar.

Por último tenemos la pila, esta se utilizará en todas las subrutinas que encontremos a lo largo del programa para asegurarnos de que el código está encapsulado y que la ejecución de las subrutinas no chafa valores del programa invocador, de igual forma nos permitirá guardar el valor del pc al que se debe retornar una vez se finalice la subrutina.

A continuación se muestra el bloque de activación de la pila que utilizaremos a lo largo de la ejecución del programa. La cual comenzará en la dirección 0X400000A0. Tras analizar la ejecución del código se observa que en este programa la pila simplemente guarda el valor de retorno del invocador de la misma y después en caso de que sea necesario guarda los parámetros fundamentales del programa invocador que no se deben perder. Esta estructura se repite en el caso de las subrutinas recursivas, en las cuales gracias a la actualización del registro r13 (puntero a la pila) se permite la apilación de los registros necesarios para volver al contexto que se tenía anteriormente ya que cada subrutina apila a partir de la dirección de sp sin chafar la información apilada por otras subrutinas.




















1.5 Comparación marco de activación de la pila

Las mayores diferencias observadas tras el análisis de la pila de activación que se nos proporciono es la utilización del registro r13 para almacenar la dirección del último valor de la pila, dado que en la versión de la pila proporcionada en la asignatura de aoc1 se utilizaba el fp en su lugar.

Además destaca el uso de los registros r0-r3 dado que estos se pueden utilizar para el paso de parámetros entre funciones, lo cual hasta el momento se había realizado a través de la pila solamente.

1.6 Análisis del rendimiento

Module/Function	Calls	Time(Sec)	Time(%)	
conecta4_jugar	1	5.249 s	67%	
entrada_nueva	4468267	2.606 s	33%	
conecta4_buscar_alineamiento_c	34	208.917 us	0%	
conecta4_hay_linea_c_c	3	77.750 us	0%	
__scatterload_zeroinit	1	59.750 us	0%	
C4_fila_valida	38	29.917 us	0%	
C4_columna_valida	33	26.833 us	0%	
C4_calcular_fila	4	18.833 us	0%	
celda_vacia	37	18.500 us	0%	
C4_comprobar_empate	2	8.167 us	0%	
C4_verificar_4_en_linea	3	7.750 us	0%	
C4_actualizar_tablero	3	5.750 us	0%	
__scatterload_copy	4	5.500 us	0%	
__user_setup_stackheap	1	4.250 us	0%	
__scatterload_rt2	1	3.750 us	0%	
entrada_leer	4	2.333 us	0%	
entrada_inicializar	3	2.000 us	0%	

Las funciones que más tiempo consumen en la ejecución de una partida, son aquellas que esperan la introducción de los datos por parte del usuario, esto se debe a la lentitud de los mismos respecto a la velocidad del procesador. Lo cual se observa en las llamadas que se ha realizado a esta función 4468267 en el caso de esta prueba y el porcentaje que representa llegando a ser el 33% de la ejecución del programa. Aislando las funciones que esperan la introducción de datos para poder continuar con la ejecución del juego podemos observar que las funciones que más tiempo de ejecución consumen son *conecta4_buscar_alineamiento_c* y *conecta4_hay_linea_c_c* seguidas de funciones como *C4_columna_valida* y *C4_fila_valida* las cuales se invocan desde las funciones anteriores. En conclusión a nivel computacional las funciones que más tiempo ocupan en la ejecución del programa son *conecta4_buscar_alineamiento_c* y *conecta4_hay_linea_c_c* las cuales se implementarán posteriormente en lenguaje ensamblador para estudiar la variación del tiempo de ejecución en diversas implementaciones.

1.7 Programación de funciones en ARM

Con el objetivo de familiarizarse con el código en lenguaje ensamblador, y aprender sobre el funcionamiento del mismo, y como los lenguajes de programación de alto nivel son traducidos a bajo nivel por el compilador, se van a proceder a implementar diversas versiones del código proporcionado para la realización de la práctica, se van a implementar las siguientes funciones:

- *conecta4_buscar_alineamiento_arm*
- *conecta4_hay_linea_arm_c*
- *conecta4_hay_linea_arm_arm*

1.7.1 conecta4_buscar_alineamiento_arm

El objetivo es implementar una función equivalente a la de c, para ello se creará la función *conecta4_hay_linea_c_arm* en la que se invocará a *conecta4_buscar_alineamiento_arm*.

De esta forma ya podremos ejecutar nuestro programa con la nueva función. A continuación se muestra el código de la función implementada.

```
1  conecta4_buscar_alineamiento_arm
2      STMDB R13!, {R4-R10,R14}
3      mov r4, r0                ; r4 = tablero
4      mov r5, r1                ; r5 = fila
5      mov r6, r2                ; r6 = columna
6      mov r7, r3                ; r7 = color
7      ; comprobamos si son correctos los valores de la celda que nos proporcionan
8      ; estan dentro del tablero
9      cmp r5, #1
10     movlt r0, #0
11     blt termina1              ; salta si r5 < 1
12     cmp r5, #6
13     movgt r0, #0
14     bgt termina1              ; salta si r5 > NUM_FILAS
15     cmp r6, #1
16     movlt r0, #0
17     blt termina1              ; salta si r6 < 1
18     cmp r6, #7
19     movgt r0, #0
20     bgt termina1              ; salta si r6 > NUM_COLUMNAS
21     ; comprobamos que la celda no sea vacia, y sea del mismo color
22     add r10, r4, r5, LSL #3    ; r10 = @tablero + 8*fila
23     ldrb r9, [r10, r6]        ; r9 = dato de la celda = r10 + columna
24     tst r9, #0x4              ; and lógico que actualiza los flags
25     moveq r0, #0
26     beq termina1              ; salta si flag z = 1 pq la celda estará vacia
27     and r10, r9, #0x03        ; and logico para encontrar color de la celda
28     cmp r10, r7                ; comparacion del color obtenido con el guardado en r7
29     movne r0, #0
30     bne termina1              ; salta si no son iguales
31     ; obtenemos el valor de delta y lo avanzamos
32     ldr r9, [sp, #32]          ; r9 = deltaFila = sp + 12
33     add r1, r5, r9              ; r5 = nueva_fila = fila + delta_fila
34     ldr r10, [sp, #36]         ; r10 = deltaColumna = sp + 8
35     add r2, r6, r10            ; r6 = nueva_columna = columna + delta_columna
36     ;llamamos a la función recursiva
37     mov r8, #1                ; apilamos en resultado un 1
38     STMDB R13!, {r8}
39     STMDB R13!, {r9, r10}      ; apilamos los parámetros y el resultado para la siguiente iteración
40     bl conecta4_buscar_alineamiento_arm
41     ldr r1, [sp, #8]           ; leemos resultado
42     add r0, r0, r1              ; r0 = resultado final
43     add sp, sp, #12            ; liberamo el esacio de los parámetros apilados
44 termina1 ; salimos de la subrutina
45     LDMIA R13!, {R4-R10, PC}
46     END
```

A esta función se le van a pasar los siguientes parámetros: (a la izquierda aparece el registro a través del cual se envían los parámetros)

- r0 = dirección del tablero
- r1 = fila seleccionada
- r2 = columna seleccionada
- r3 = color

Por último los parámetros de delta fila, y delta columna se pasarán como parámetro a través de la pila, por lo que se deberá tener en consideración la posición en la pila que ocupan para poder obtener estos valores. Siendo en nuestra implementación

- delta fila $sp + 12$
- delta columna $sp + 8$

Además se ha tenido en consideración que dado que se utiliza el estándar ATPCS el resultado se deberá devolver en el registro r0, que es donde lo buscará posteriormente la función invocadora en c.

Dado que esta función es recursiva ha realizado un estudio de la pila que nos pasaba el programa invocador, dado que las posiciones de los parámetros obtenidos a través de la pila deberá ser la misma que en las llamadas recursivas.

La función se divide en tres bloques principales, en el primero de ellos (líneas 1-31) se encarga de guardar el contexto para que no existan problemas posteriormente con el resto de funciones. Además nos encargamos de realizar las comprobaciones correspondientes para saber si es la última invocación (es decir si estamos en el caso base) si se cumple esto la invocación recursiva terminará y empezaremos a recuperar los valores de las primeras invocaciones. Para ello se salta al bloque 3.

El segundo apartado (32-40) se encarga de preparar los valores para la siguiente invocación (este paso se salta si se ha entrado en un caso base). Para ello se actualizan los deltas y se apilan para la siguiente invocación.

En el tercer bloque (41-45) se ejecuta a la vuelta de las llamadas recursivas, actualizando el resultado con el valor proporcionado por todas las subrutinas, se devuelven los parámetros a sus valores correspondientes que se habían almacenado al principio de la ejecución y se recupera el valor de pc del programa invocador.

Por último vamos a proceder a explicar la utilización que se le ha dado a cada uno de los registros en la ejecución de los programas:

- r0 = r4 = @ de tablero y paso de resultados a la función invocadora
- r1 = r5 = fila
- r2 = r6 = columna
- r3 = r7 = color de la ficha
- r8 = se utilizará para guardar el valor #1 para apilarlo como resultado en caso de que sea necesario.
- r9 = deltaFila
- r10 = deltaColumna

1.7.2 conecta4_hay_linea_arm_c

Esta función llamará a conecta4_buscar_alineamiento_c, por ello lo que se encargará de gestionar es la llamada a esta función, y de invocarla con los parámetros adecuados para estudiar todos los casos que se están dando en el tablero actual (en horizontal, en vertical y en diagonal). A continuación se muestra el código de la función:



```
1  conecta4_hay_linea_arm_c
2
3      STMDB R13!, {R4-R12, R14}
4      mov r4, #0                ; contador i del bucle for
5      mov r6, r0                ; valor auxiliar del tablero
6      mov r5, r1                ; r5 = valor de la fila
7      mov r8, r2                ; r8 = valor de la columna
8      mov r12, r3               ; r12 = color
9      LDR r7, =deltas_fila      ; r7 = @deltas_fila
10
11  for1
12      ; actualizamos los deltas
13      ldrsb r9, [r7]            ; r9 = valor deltas_fila
14      ldrsb r10, [r7, #4]       ; r10 = valor deltas_columna; r8 = r8 + 1
15      add r7, r7, #1            ; r7 = r7 + 1
16      STMDB R13!, {r9,r10}      ; apilamos los deltas
17      ; invocamos a la función para buscar en un sentido
18      bl conecta4_buscar_alineamiento_c
19      add sp, sp, #8            ; liberamos los parámetros
20      cmp r0, #4                ; salta si r4 >= 4
21      movge r11, r0
22      bge continua1
23      ; preparamos los parametros para la siguiente invocación
24      mov r11, #-1              ; r12 = -1 para actualizar los deltas
25      mov r1, r5                ; devolvemos el valor a r1 (fila)
26      mov r2, r8                ; devolvemos el valor a r2 (columna)
27      sub r1, r1, r9            ; fila = fila - delta_fila
28      sub r2, r2, r10           ; columna = columna - delta_columna
29      mul r9, r11, r9
30      mul r10, r11, r10
31      ; devolvemos los parámetros a sus registros correspondientes
32      mov r11, r0                ; resultado temporal en r11
33      mov r0, r6                ; devolvemos el valor a r0 (cuadrícula)
34      mov r3, r12               ; devolvemos el valor a r3 (color)
35      STMDB R13!, {r9,r10}      ; apilamos los deltas
36      bl conecta4_buscar_alineamiento_c
37      add r11, r11, r0          ; guardamos en r11 el valor del resultado actualizado
38      add sp, sp, #8            ; liberamos los parámetros
39      cmp r11, #4                ; salta si r4 >= 4
40      bge continua1
41      ; devolvemos a la normalidad los registros para la siguiente iteración
42      mov r0, r6                ; devolvemos el valor a r0 (cuadrícula)
43      mov r1, r5                ; devolvemos el valor a r1 (fila)
44      mov r2, r8                ; devolvemos el valor a r2 (columna)
45      mov r3, r12               ; devolvemos el valor a r3 (color)
46      ; comprobamos si volvemos a saltar al bucle
47      add r4, r4, #1            ; incrementamos contador
48      cmp r4, #4                ; salta si r4 < 4
49      blt for1
50      ; salimos del bucle y terminamos la subrutina
51  continua1
52      cmp r11, #4                ; Guardamos el resultado en r0 para devolverlo
53      movge r0, #1
54      movlt r0, #0
55      LDMIA R13!, {R4-R12,PC}
```

A esta función se le van a pasar los siguientes parámetros:

- r0 = dirección del tablero
- r1 = fila seleccionada
- r2 = columna seleccionada
- r3 = color

En este caso dado que los deltas se definen en el propio fichero (dado que se crean en la propia función). Además no se trata de una función recursiva como las anteriores, si no que se compone de un bucle y dos invocaciones a la función `conecta4:hay_linea_c`.

En esta función las divisiones del código son las siguientes:

Las Líneas 1-16 se encargan de preparar los registros y guardar los datos que son necesarios guardar para usarlos posteriormente y prepara la pila para la invocación de la siguiente función.

18-22 y 37-40 comprobamos los resultados obtenidos de la invocación y liberamos el espacio de la pila, si es necesario se termina la ejecución del programa (caso de que se devuelva un valor igual a 4 o superior).

23-35 se recupera el valor de los registros necesarios, y se modifican los parámetros para realizar la segunda llamada a la función, la cual evaluará el tablero en sentido inverso a la primera invocación.

41-49 se prepara los registros para la siguiente iteración del bucle en la que se estudiará una sección diversa del tablero, en caso de que el resultado de la suma de ambas invocaciones sea 4 o superior se saldrá del bucle, y se ira a la última sección del código (50-55) en la que se colocará el valor correspondiente en r0, recordemos que esta función devuelve un booleano y devolverá los registros a sus correspondientes valores guardados al principio de la ejecución.

Utilización de los registros:

- r0 = @tablero y resultados
- r1 = fila
- r2 = columna
- r3 = color
- r4 = contador i para el bucle for
- r9 = valor delta_fila
- r10 = valor delta_columna
- r11 = -1 (valor para actualizar los deltas), y resultados temporales

Estos cuatro registros actúan como variables auxiliares dado que su valor se modifica en las invocaciones a función.

- r5 = fila
- r6 = @tablero
- r7 = @deltas
- r8 = columna

1.7.3 conecta4_hay_linea_arm_arm

En este apartado se va a explicar la función `conecta4_hay_linea_arm_arm`. Esta función programada en ARM va a unificar las funciones de `conecta4_hay_linea_arm_c` y `conecta4_buscar_alineamiento_arm` eliminando así la recursividad en el código y mejorando su tiempo de ejecución.



```
1  conecta4_hay_linea_arm_arm
2      STMDB R13!, {R4-R12, R14}
3      mov r4, #0          ; contador i del bucle for
4      LDR r7, =deltas_fila ; r7 = @deltas_fila
5  for
6      mov r8, #0          ; contador resultado para alineamiento
7      ; funcion alineamiento
8      ; actualizamos los deltas
9      ldrsb r6, [r7]      ; r4 = valor deltas_fila
10     ldrsb r5, [r7, #4]   ; r5 = valor deltas_columna; r8 = r8 +1
11     ; comprobamos si son correctos los valores de la celda que nos proporcionan
12     ; estan dentro del tablero
13     mov r11, r1
14     mov r12, r2
15  buc cmp r1, #1
16     blt termina          ; salta si r1 < 1
17     cmp r1, #6
18     bgt termina          ; salta si r5 > NUM_FILAS
19     cmp r2, #1
20     blt termina          ; salta si r6 < 1
21     cmp r2, #7
22     bgt termina          ; salta si r6 > NUM_COLUMNAS
23     ; comprobamos que la celda no sea vacia, y sea del mismo color
24     add r10, r0, r1, LSL #3 ; r10 = @tablero + 8*fila
25     ldrb r9, [r10, r2]    ; r9 = dato de la celda = r10 + columna
26     tst r9, #0x4         ; and lógico que actualiza los flags
27     beq termina          ; salta si flag z = 1 pq la celda estará vacia
28     and r10, r9, #0x03    ; and logico para encontrar color de la celda
29     cmp r10, r3           ; comparacion del color obtenido con el guardado en r7
30     bne termina
31     add r1, r1, r6        ; nueva_fila = fila + delta_fila
32     add r2, r2, r5        ; nueva_columna = columna + delta_columna
33     add r8, r8, #1        ; incremento resultado
34     b buc
35  termina
36     cmp r8, #4            ; salta si r4 >= 4
37     bge continua
38     mov r1, r11
39     mov r2, r12
40     ; preparamos los parametros para la siguiente invocación
41     ldrsb r6, [r7]        ; r4 = valor deltas_fila
42     ldrsb r5, [r7, #4]    ; r5 = valor deltas_columna; r8 = r8 +1
43
44     mov r10, #-1          ; r10 = -1 para actualizar los deltas
45     sub r1, r1, r6        ; fila - delta_fila
46     sub r2, r2, r5        ; columna - delta_columna
47     mul r6, r10, r6       ; delta_fila
48     mul r5, r10, r5       ; delta_columna
```

```

49      ; comprobamos si son correctos los valores de la celda que nos proporcionan
50      ; estan dentro del tablero
51  buc1
52      cmp r1, #1
53      blt term          ; salta si r1 < 1
54      cmp r1, #6
55      bgt term          ; salta si r5 > NUM_FILAS
56      cmp r2, #1
57      blt term          ; salta si r6 < 1
58      cmp r2, #7
59      bgt term          ; salta si r6 > NUM_COLUMNAS
60      ; comprobamos que la celda no sea vacia, y sea del mismo color
61      add r10, r0, r1, LSL #3      ; r10 = @tablero + 8*fila
62      ldrb r9, [r10, r2]          ; r9 = dato de la celda = r10 + columna
63      tst r9, #0x4                ; and lógico que actualiza los flags
64      beq term                    ; salta si flag z = 1 pq la celda estará vacia
65      and r10, r9, #0x03          ; and logico para encontrar color de la celda
66      cmp r10, r3                ; comparacion del color obtenido con el guardado en r7
67      bne term
68      add r1, r1, r6              ; nueva_fila = fila + delta_fila
69      add r2, r2, r5              ; nueva_columna = columna + delta_columna
70      add r8, r8, #1              ; incremento resultado
71      b buc1
72  term
73      cmp r8, #4                  ; salta si r4 >= 4
74      bge continua
75      add r7, r7, #1              ; r7 = r7 +1
76      add r4, r4, #1              ; incrementamos contador
77      cmp r4, #4                  ; salta si r4 < 4
78      mov r1, r11
79      mov r2, r12
80      blt for
81      ; salimos del bucle y terminamos la subrutina
82  continua
83      cmp r8, #4                  ; Guardamos el resultado en r0 para devolverlo
84      movge r0, #1
85      movlt r0, #0
86      LDMIA R13!, {R4-R12,PC}

```

Al igual que en la función anterior se le van a pasar los siguientes registros:

- r0 = dirección del tablero
- r1 = fila seleccionada
- r2 = columna seleccionada
- r3 = color

Debido a la limitación del número de registros que se tiene se ha optado por realizar un código de mayor tamaño para poder cumplir con el estándar y no utilizar registros que pueden resultar conflictivos.

La estructura que se ha seguido si se observa el código es la misma que en las dos funciones previamente explicadas con la principal diferencia de que se ha eliminado la recursividad sustituyéndola por bucles etiquetados como buc y buc1.

Registros utilizados:

- r0 = @tablero
- r1 = fila
- r2 = columna
- r3 = color
- r4 = contador del bucle exterior (el for en c)
- r5 = delta_columna
- r6 = delta_fila
- r7 = @deltas
- r8 = resultado temporal
- r9 y r10 = se utilizan para calcular la @ de la celda
- r11 = copia de fila
- r12 = copia de columna

1.8 Verificación automática y comparación de los resultados

Para la verificación del correcto funcionamiento de cada una de las versiones del código anteriormente mencionado se ha realizado un fichero de pruebas llamado “pruebas.c” en el cual se ejecutan jugadas en 6 tableros distintos comprobando diversas situaciones que se han considerado relevantes.

Además cada uno de los resultados obtenidos por las funciones se compara comprobando que el resultado proporcionado por todas ellas es el mismo. Utilizando el programa proporcionado en c como guía para la verificación de los resultados.

A continuación se muestran los tableros utilizados en las diversas pruebas:

```

1 static CELDA
2 cuadrricula_color[7][8] =
3 {
4     0, 0XC1, 0XC2, 0XC3, 0XC4, 0XC5, 0XC6, 0XC7,
5     0XF1, 6, 5, 6, 5, 6, 6, 6,
6     0XF2, 5, 6, 5, 5, 6, 5, 6,
7     0XF3, 6, 6, 5, 6, 6, 5, 5,
8     0XF4, 5, 6, 5, 6, 5, 5, 6,
9     0XF5, 6, 5, 6, 5, 6, 6, 5,
10    0XF6, 5, 5, 6, 6, 6, 5, 0};

```

Comprobamos el caso de empate

```

1 static CELDA
2 cuadrricula_columna[7][8] =
3 {
4     0, 0XC1, 0XC2, 0XC3, 0XC4, 0XC5, 0XC6, 0XC7,
5     0XF1, 6, 5, 6, 5, 6, 6, 6,
6     0XF2, 5, 6, 5, 5, 6, 5, 6,
7     0XF3, 6, 6, 5, 6, 6, 5, 5,
8     0XF4, 0, 6, 5, 6, 5, 5, 6,
9     0XF5, 0, 5, 6, 5, 6, 6, 5,
10    0XF6, 0, 5, 6, 6, 6, 5, 6};

```

Aseguramos correcta comprobación de columna válida

```

1 static CELDA
2 cuadrricula_fila[7][8] =
3 {
4     0, 0XC1, 0XC2, 0XC3, 0XC4, 0XC5, 0XC6, 0XC7,
5     0XF1, 6, 5, 0, 5, 6, 6, 6,
6     0XF2, 5, 6, 0, 5, 6, 5, 6,
7     0XF3, 6, 6, 0, 6, 6, 5, 5,
8     0XF4, 6, 6, 0, 6, 5, 5, 6,
9     0XF5, 5, 5, 0, 5, 6, 6, 5,
10    0XF6, 5, 5, 0, 6, 6, 5, 6};

```

Comprobación fila válida y celda vacía

```

1 static CELDA
2 cuadrícula_victoria_di[7][8] =
3 {
4     0, 0XC1, 0XC2, 0XC3, 0XC4, 0XC5, 0Xc6, 0XC7,
5     0XF1, 6, 5, 6, 5, 6, 6, 6,
6     0XF2, 5, 6, 5, 5, 6, 5, 0,
7     0XF3, 6, 6, 5, 6, 6, 5, 0,
8     0XF4, 5, 6, 5, 6, 5, 5, 0,
9     0XF5, 6, 5, 6, 5, 6, 6, 0,
10    0XF6, 5, 5, 6, 6, 6, 5, 0};

```

Comprobación caso de victoria diagonal 1

```

1 static CELDA
2 cuadrícula_victoria_dd[7][8] =
3 {
4     0, 0XC1, 0XC2, 0XC3, 0XC4, 0XC5, 0Xc6, 0XC7,
5     0XF1, 6, 5, 6, 5, 6, 6, 6,
6     0XF2, 5, 6, 0, 5, 6, 5, 0,
7     0XF3, 6, 6, 0, 5, 6, 5, 0,
8     0XF4, 5, 6, 0, 6, 5, 5, 0,
9     0XF5, 6, 5, 0, 5, 6, 6, 0,
10    0XF6, 5, 5, 0, 6, 6, 5, 0};

```

Comprobación caso de victoria diagonal 2

```

1 static CELDA
2 cuadrícula_victoria_v[7][8] =
3 {
4     0, 0XC1, 0XC2, 0XC3, 0XC4, 0XC5, 0Xc6, 0XC7,
5     0XF1, 6, 5, 6, 5, 6, 6, 5,
6     0XF2, 5, 6, 0, 5, 6, 5, 5,
7     0XF3, 6, 6, 0, 5, 6, 6, 5,
8     0XF4, 5, 6, 0, 6, 5, 5, 0,
9     0XF5, 6, 5, 0, 5, 6, 6, 0,
10    0XF6, 5, 5, 0, 6, 6, 5, 0};

```

Comprobación caso de victoria vertical

Para la realización de estas pruebas se ha cambiado el método de introducción de casillas automatizandolo. Por lo que el propio código del fichero pruebas realiza todas los movimientos de forma autónoma. Acelerando y facilitando así el proceso de pruebas.

2 Resultados

2.1 Medidas de rendimiento

En esta sección del trabajo se va a proceder a tomar las medidas de rendimiento de las versiones implementadas para ello en la siguiente tabla se muestran tanto el tamaño en bytes de cada función como su tiempo de ejecución para cada versión.

Versión	Funciones que utiliza	Tamaño (Bytes)	Tiempo (us)
C-C	conecta4_hay_linea_c_c	304	14,583
	conecta4_buscar_alineamiento_c	156	66,917
ARM-C	conecta4_hay_linea_arm_c	168	8,583
	conecta4_buscar_alineamiento_c	156	66,917

C-ARM	conecta4_hay_linea_c_arm	304	14,583
	conecta4_buscar_alineamiento_arm	148	50,750
ARM-ARM	conecta4_hay_linea_arm_arm	272	22,917

Como podemos observar el paso de las funciones escritas en alto nivel a bajo nivel (ARM) mejora considerablemente tanto los resultados en tiempo de ejecución, como el número de bytes que ocupa cada función, mostrando así cómo este nuevo código acelera la ejecución del algoritmo respecto al proporcionado en C (teniendo en cuenta que no se está realizando ninguna optimización por parte del compilador); siendo la función ARM-ARM la que ha obtenido mejores resultados en tiempo de ejecución y tamaño en bytes de todas las versiones.

2.2 Optimizaciones del compilador

Inicialmente las mediciones se han realizado con la optimización -O0 cuyos resultados se muestran en el apartado anterior. Ahora se va a pasar a realizar las mismas mediciones pero utilizando los distintos niveles de optimización que nos proporciona el compilador para estudiar los resultados obtenidos, los cuales se recogen en la siguiente tablas de las optimizaciones restantes.

Optimizacion -O1:

Versión	Funciones que utiliza	Tamaño (Bytes)	Tiempo (us)
C-C	conecta4_hay_linea_c_c	256	12,667
	conecta4_buscar_alineamiento_c	184	51,083
ARM-C	conecta4_hay_linea_arm_c	168	8,583
	conecta4_buscar_alineamiento_c	184	66,917
C-ARM	conecta4_hay_linea_c_arm	256	12,667
	conecta4_buscar_alineamiento_arm	148	50,750
ARM-ARM	conecta4_hay_linea_arm_arm	272	22,917

Optimizacion -O2:

Versión	Funciones que utiliza	Tamaño (Bytes)	Tiempo (us)
C-C	conecta4_hay_linea_c_c	148	11,667
	conecta4_buscar_alineamiento_c	108	36,417

ARM-C	conecta4_hay_linea_arm_c	168	8,583
	conecta4_buscar_alineamiento_c	108	36,417
C-ARM	conecta4_hay_linea_c_arm	148	11,667
	conecta4_buscar_alineamiento_arm	148	50,750
ARM-ARM	conecta4_hay_linea_arm_arm	272	22,917

Optimizacion -O3:

Versión	Funciones que utiliza	Tamaño (Bytes)	Tiempo (us)
C-C	conecta4_hay_linea_c_c	400	16,917
	conecta4_buscar_alineamiento_c	108	25,833
ARM-C	conecta4_hay_linea_arm_c	168	8,583
	conecta4_buscar_alineamiento_c	108	36,417
C-ARM	conecta4_hay_linea_c_arm	228	0
	conecta4_buscar_alineamiento_arm	148	50,750
ARM-ARM	conecta4_hay_linea_arm_arm	272	22,917

Tras la obtención de estos resultados podemos observar que la optimización -O0 es la que produce el código más ineficiente, en el cual se realizan muchas más llamadas a funciones y memoria, y por ende el tiempo de ejecución es mayor que en otras optimizaciones.

Además, en la mayoría de los casos cada vez que se aplica una optimización mayor se mejora el tiempo de ejecución de las funciones respecto de las optimizaciones anteriores. Esto también sucede con el tamaño en bytes de cada función, el cual por regla general va disminuyendo según se va subiendo el nivel de optimización, aunque esto no siempre es así; en el caso de la función *conecta4_hay_linea_c_c* su tamaño en bytes aumenta con el nivel de optimización -O3, ya que el compilador trabaja y ordena el código producido en bajo nivel según cree que es la forma más optima de organización.

Las únicas funciones en las cuales no cambia ni el tamaño de la función, ni el tiempo de ejecución son aquellas programadas en lenguaje ensamblador, esto es debido a que los cambios en el nivel de optimización afectan principalmente a las funciones de alto nivel escritas en lenguaje C.

Otro aspecto importante a comentar el caso de la función *conecta4_hay_linea_c_arm* , como se puede apreciar en la tabla dedicada al nivel de optimización -O3, el tiempo de ejecución de esta función se reduce a 0us, esto es debido a que la optimización evita hacer una llamada a esa función, reduciendo así los saltos, pasos de parámetros y otras interacciones que aumentaban el tiempo de ejecución total.

En conclusión, se puede apreciar que cada optimización (-O0, -O1, -O2, -O3) en términos generales ofrece una mejora tanto en tamaño en bytes como en tiempo de ejecución siendo así en términos de rendimiento el nivel de optimización -O3 el que ofrece una mejora más significativa, con un tiempo de ejecución mucho menor que el obtenido con las demás optimizaciones.

3 Conclusiones

Tras la realización de la práctica se ha extraído la conclusión de que el código en ensamblador directo tiende a ser más óptimo que el que se ha realizado a alto nivel siempre que el compilador no aplique ningún nivel de optimización posteriormente al código en alto nivel.

De igual forma se ha concluido que los diversos niveles de optimización eliminan y reorganizan el código llegando a obtener buenos resultados, siendo el nivel de optimización -O3 el que proporciona una mejora más significativa respecto de los demás.

Algunas de las diversas dificultades que han surgido en el proceso de elaboración de la práctica consisten en la gestión de registros dado que el número de los registros de los que se disponen es muy reducido, o bien en adquirir los conocimientos necesarios sobre el estándar ATPCS y cómo las funciones en c hacen uso del mismo y pasan los parámetros y recogen el resultado de registros concretos los cuales no se deben modificar.

Se considera que se han cumplido los objetivos establecidos de mejorar el nivel de comprensión de código ensamblador, al igual que los conocimientos respecto al compilador y sus diversos niveles de optimización.