Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

proyhw-pr-1

Proyecto Hardware

Autor 1:	Toral Pallás, Héctor - 798095
Autor 2:	Lahoz Bernad, Fernando - 800989
Grado:	Ingeniería Informática
Curso:	2022-2023

20 de Septiembre de 2022

Índice

1. Resumen

Entre los principales objetivos de este proyecto se encuentra reforzar el dominio del lenguaje de programación C en relación a cómo se traducen distintas estructuras de código al lenguaje ensamblador y qué técnicas se puede emplear para optimizar su funcionamiento. Para ello se ha procedido a traducir las dos funciones del código del juego Conecta 4 en las que se concentra la mayor parte del tiempo de ejecución, siguiendo el estándar ATPCS. A través de diferentes tests y mediciones tanto de tiempo como de espacio en memoria se concluye que el comportamiento de las funciones es equivalente al código original, que es más eficiente en tiempo y que además el código ocupa menos espacio en memoria.

Para el desarrollo del proyecto ha sido necesario estudiar el funcionamiento de la arquitectura ARM, en concreto del microcontrolador LPC2105, y su ejecución segmentada en 3 etapas. También requería tener clara una idea del mapa de memoria, la función de cada zona y su tamaño. La depuración del código, las distintas ejecuciones y las mediciones han sido realizadas en el entorno de desarrollo Keil µVision.

2. Memoria y Pila

2.1. Mapa de memoria

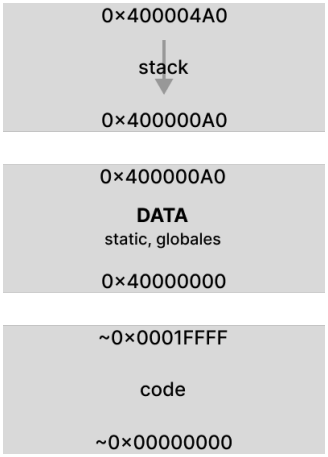


Figura 1: Mapa de memoria

Stack: 1MiB de memoria que sirve de estructura LIFO, *full-descending*, donde se guardan las variables de ámbito y el estado del programa antes de invocar a una función.

Data: 160B de memoria donde se almacenan las variables globales y definidas con el modificador *static*.

Code: Área de sólo lectura que almacena las instrucciones en código máquina que se van a ejecutar. Los valores inmediatos que por tamaño no se pueden incluir en el código de la instrucción, como direcciones de memoria de las variables y otras constantes, se almacenan aquí y se accede a ellos a través de PC.

2.2. Bloque de activación

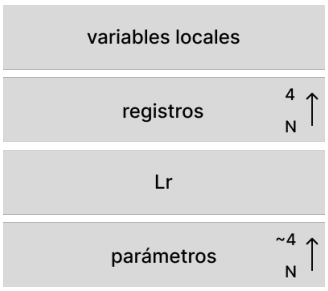


Figura 2: Bloque de activación

Variables locales: Si alguno de los parámetros pasados por r0-r3 es *"volatile"*, lo guarda r0-r3 aquí antes que cualquier otra variable.

Registros: Aquí se almacenan aquellos que van a ser utilizados, primero el registro de mayor ordinal, de tal manera que podamos recuperar su valor al retornar.

LR: Almacena la dirección de retorno (pc+4). Sólo se guarda si dentro de la función se llama a otra.

Parámetros: En esta zona se almacenan aquellos parámetros que no tengan espacio en los registros r0-r3. Los parámetros se almacenan en los tres primeros registros de izquierda a derecha con las siguientes normas:

- Si el parámetros ocupa menos de 32bits se extiende.
- Si el parámetro ocupa más de 32bits se usa más de un registro para pasarlo.
- Si el valor devuelto es un struct de más de 64bits, el registro r0 contiene la dirección de memoria del struct a sobreescribir.

En la asignatura Arquitetura y Organización de Computadores se vió un bloque de activación similar, con la diferencia de que siempre se reservaba espacio para el valor de retorno y para todos los parámetros. Además, en el modelo de rutinas aprendido siempre se hacía uso del frame pointer para el acceso a los parámetros, por lo que se siempre se guardaba junto con r14, y las variables locales se apilaban antes de guardar el valor de los registros. En el marco de pila visto en las diapositivas del proyecto se almacena también PC.

3. Funciones refactorizadas e implementadas

```
  AREA codigo, CODE
IMPORT conecta4_buscar_alineamiento_c
EXPORT conecta4_buscar_alineamiento_arm
EXPORT conecta4_hay_linea_arm_c
EXPORT conecta4_hay_linea_arm_arm
PRESERVE8 {TRUE}
NUM_COLUMNAS    EQU 7
NUM_FILAS       EQU 6
  ENTRY

```

3.1. conecta4_hay_linea_c_arm

3.1.1. Código

El código de la función es idéntico al de *conecta4_hay_linea_c_c*, cambiando las llamadas a *conecta4_buscar_alineamiento_c* por la version de arm. Para evitar que el compilador haga inline de la función y medir el tiempo con fidelidad ha hecho falta incluir la orden *__attribute__((noinline))*.

```
__attribute__((noinline)) uint8_t conecta4_hay_linea_c_arm(
    CELDA cuadricula[TAM_FILS][TAM_COLS],
    uint8_t fila, uint8_t columna, uint8_t color
) {
  int8_t deltas_fila[4] = {0, -1, -1, 1};
  int8_t deltas_columna[4] = {-1, 0, -1, -1};
  unsigned int i = 0;
  uint8_t linea = FALSE;
  uint8_t long_linea = 0;

  for (i = 0; (i < 4) && (linea == FALSE); ++i) {
    long_linea = conecta4_buscar_alineamiento_arm(
        cuadricula, fila, columna, color,
        deltas_fila[i], deltas_columna[i]);

    linea = long_linea >= 4;
    if (linea) continue;

    long_linea += conecta4_buscar_alineamiento_arm(
        cuadricula, fila - deltas_fila[i], columna - deltas_columna[i],
        color, -deltas_fila[i], -deltas_columna[i]);

    linea = long_linea >= 4;
  }
  return linea;
}
```

conecta4_buscar_alineamiento_arm

r0	r1	r2	r3	mem[sp]	mem[sp+4]
celda cuadrícula	uint8_t fila	uint8_t columna	uint8_t color	int8_t delta_fila	int8_t delta_columna

```
conecta4_buscar_alineamiento_arm
  STMDB sp!, {r4, r5, lr} ; Guarda valor de los registros para utilizarlos

  ; return si fila no valida
  cmp r1, #1
  blo return0             ; salta si: fila < 1
  cmp r1, #NUM_FILAS
  bhi return0             ; salta si: fila > NUM_FILAS

  ; return si columna no valida
  cmp r2, #1
  blo return0             ; salta si: columna < 1
  cmp r2, #NUM_COLUMNAS
  bhi return0             ; salta si: columna > NUM_COLUMNAS
```

```
  ; r4 := celda
  add r4, r0, r2          ; r4 = r0+r2 = &cuadricula[0][columna]
  ldrb r4, [r4, r1, LSL#3]; r4 = mem[r4+r1*8] = cuadricula[fila][columna]
  ; return si celda vacia
  tst r4, #0x4
  beq return0             ; salta si: celda vacia

  ; return si celda != color
  and r5, r4, #0x3
  cmp r5, r3
  bne return0             ; salta si: celda != color

  ; avanzar indices
  ldr r4, [sp, #12]       ; r4 = delta_fila
  ldr r5, [sp, #16]       ; r4 = delta_columna
  add r1, r1, r4          ; fila    = fila    + delta_fila
  add r2, r2, r5          ; columna = columna + delta_columna

  ; carga de parametros
  STMDB sp!, {r4, r5}     ; delta_columnas := r5, delta_filas := r4
  bl conecta4_buscar_alineamiento_arm
  add sp, sp, #8          ; libera parametros
  add r0, r0, #1          ; r0 = resultado de funcion + 1

  LDMIA sp!, {r4, r5, lr} ; Recupera el valor de los registros
  bx lr                   ; return r0

return0
  mov r0, #0              ; r0 = 0
  LDMIA sp!, {r4, r5, lr} ; Recupera el valor de los registros
  bx lr                   ; return 0
```

3.1.2. Optimizaciones aplicadas

La clave para optimizar esta función se encuentra en reformular las condiciones de ejecución:

$$!C4\_fila\_valida(fila) = !((fila \geq 1)\ and\ (fila \leq NUM\_FILAS)) = (fila < 1)\ or\ (fila > NUM\_FILAS)$$

$$!C4\_columna\_valida(columna) = !((columna \geq 1)\ and\ (columna \leq NUM\_COLUMNAS)) = \\ = (columna < 1)\ or\ (columna > NUM\_COLUMNAS)$$

De esta forma la condición se reduce a una secuencia de comprobaciones cortocircuitadas y en cuanto una no se cumple la función ejecuta *return 0*. Como la llamada recursiva se ejecuta al final, el valor de los registros no es relevante una vez finalice, por lo que es necesario devolverles el valor que tenían antes de ejecutarla.

En la versión optimizada del compilador de C, la evaluación para comprobar si la fila o la columna es válida la ejecutaba con un and cortocircuitado mediante instrucciones predicadas, en lugar de saltos. Eso favorece casos en los que se cumple $fila \geq 1$ pero no $fila \leq NUM\_FILAS$, permitiendo ahorrar 1 ciclo al no comprobar la condición del salto. En nuestra implementación se salta inmediatamente si una condición no se cumple, ahorrando 1 ciclo si directamente no se cumple $fila \geq 1$. Para ejecuciones con varias llamadas a la función resulta más óptima la versión del compilador.

En el estándar ATPCS, el registro R12 sirve como variable temporal y permite ahorrar una lectura y una escritura en memoria en cada llamada a la función, al no tener que recuperar su valor, lo que es apreciable en este tipo de bucles recursivos. El compilador de C aprovecha mucho este registro, mientras que en nuestra implementación no ha sido utilizado.

Como curiosidad, para comprobar *celda_vacia* el compilador ejecuta la instrucción **celda & 0x4** desplazando 29 posiciones a la izquierda el bit 2 y comprobando si genera desbordamiento, en lugar de utilizar la instrucción tst.

3.2. conecta4_hay_linea_arm_c

3.2.1. Código

r0	r1	r2	r3
celda cuadrícula	uint8_t fila	uint8_t columna	uint8_t color

```
conecta4_hay_linea_arm_c
  STMDB sp!, {r4-r11, lr} ; Guarda valor de los registros para utilizarlos
  LDR r4, =0x01FFFF00     ; r4 := {0, -1, -1, 1}
  LDR r5, =0xFFFF00FF     ; r5 := {-1, 0, -1, -1}
  STMDB sp!, {r4, r5}     ; Guarda los vectores en la pila
  mov r4, #0              ; i := 0
  mov r5, #0              ; linea := FALSE
  ; Backups de los parametros
  mov r9, r0              ; r9  = cuadricula_bak := cuadricula
  mov r10, r1             ; r10 = fila_bak := fila
  mov r11, r2             ; r11 = columna_bak := columna

for
  mov r6, r3              ; r6 = color_bak := color
  ; carga de parametros
  ldrsb r7, [sp, r4]      ; r7 := deltas_fila[i]
  add sp, sp, #4
  ldrsb r8, [sp, r4]      ; r8 := deltas_columna[i]
  sub sp, sp, #4
  STMDB sp!, {r7, r8}     ; delta_columnas := r8, delta_filas := r7
  bl conecta4_buscar_alineamiento_c
  add sp, sp, #8          ; libera parametros
  mov r3, r6              ; color := color_bak
  mov r6, r0              ; r6 := longLinea   // resultado de funcion
  mov r0, r9              ; r0 := cuadricula_bak

  ; return true si longLinea >= 4
  cmp r6, #4
  movhs r0, #1
  bhs return              ; salta si: longLinea >= 4

  mov r5, r3              ; r5 = color_bak := color
  ; carga de parametros
  sub r1, r10, r7         ; fila := fila + deltas_fila[i]
  sub r2, r11, r8         ; columna := columna + deltas_columna[i]
  rsb r7, r7, #0          ; r7 := -deltas_fila[i]
  rsb r8, r8, #0          ; r8 := -deltas_columna[i]
  STMDB sp!, {r7, r8}     ; delta_columnas := r8, delta_filas := r7
  bl conecta4_buscar_alineamiento_c
  add sp, sp, #8          ; libera parametros
  mov r3, r5              ; color := color_bak
  add r6, r6, r0          ; longLinea := longLinea + resultado de funcion

  cmp r6, #4
  movhs r5, #1
  movlo r5, #0            ; linea := longLinea >= 4

  ; Recupera el valor de los parametros
  mov r0, r9              ; cuadricula := cuadricula_bak
  mov r1, r10             ; fila := fila_bak
  mov r2, r11             ; columna := columna_bak

  ; guarda del bucle
  add r4, r4, #1          ; i++
  cmp r4, #4
  cmpne r5, #1
  bne for                 ; repite si: i != 4 && !linea

  mov r0, r5              ; r0 := linea   // resultado de la funcion
return
  add sp, sp, #8          ; libera vectores de pila
  LDMIA sp!, {r4-r11, lr} ; Recupera el valor de los registros
  bx lr                   ; return linea
```

3.2.2. Optimizaciones aplicadas

La principal optimización corresponde a la estructura del bucle for. No es necesario comprobar la guarda del bucle for en la primera iteración porque siempre se ejecuta. Esto permite ahorrar un salto incondicional (traducción literal) o ahorrarnos duplicar la guarda al final del bucle, y es una de las optimizaciones que realiza el compilador de C. Además la condición que se comprueba es i *!=* 4, ya que por el incremento unitario es equivalente a $i < 4$ y permite realizar un and lógico sin saltos (líneas 55 y 56).

La recuperación de los valores de los parámetros se realiza por medio de registros, para minimizar el acceso a memoria. Sin embargo, no hay suficientes registros disponibles para recuperar el parámetro *color* (r3). El valor de *long_linea* no es relevante justo antes de llamar a la primera función *buscar_alineamiento* y se sobreescribe al finalizar (por eso también se ha obviado la inicialización), por lo que se utiliza temporalmente r6 como backup de *color* en lugar de guardar su valor habitual como *long_linea*. La variable *linea* sobreescribe su valor después de cada llamada a la función. Por ello también se ha utilizado como backup de r3 en la segunda llamada a *buscar_alineamiento*.

Por último, la acción *continue* después de comprobar el valor de *linea* es equivalente a *return true*, porque al ejecutar la guarda del bucle se está repitiendo la misma comprobación.

3.3. conecta4_hay_linea_arm_arm

3.3.1. Código

r0	r1	r2	r3
celda cuadrícula	uint8_t fila	uint8_t columna	uint8_t color

```
conecta4_hay_linea_arm_arm
  STMDB sp!, {r4-r11, lr} ; Guarda valor de los registros para utilizarlos
  LDR r4, =0x01FFFF00     ; r4 = {0 , -1 , -1 , 1}
  LDR r5, =0xFFFF00FF     ; r5 = { -1 , 0 , -1 , -1}
  STMDB sp!, {r4, r5}     ; Guarda los vectores en la pila
  mov r6, #0              ; i := 0
  mov r10, r1             ; r10 := fila
  mov r11, r2             ; r11 := columna

  ; return si fila no valida
  cmp r1, #1
  blo returnFalse         ; salta si : fila < 1
  cmp r1, #NUM_FILAS
  bhi returnFalse         ; salta si : fila > NUM_FILAS

  ; return si columna no valida
  cmp r2, #1
  blo returnFalse         ; salta si : columna < 1
  cmp r2, #NUM_COLUMNAS
  bhi returnFalse         ; salta si : columna > NUM_COLUMNAS

  ; r4 := celda
  add r4, r0, r2          ; r4 = r0 + r2 = & cuadricula [0][columna]
  ldrb r4, [r4, r1, LSL#3]; r4 = mem[r4+r1*8] = cuadricula [fila][columna]
  ; return si celda vacia
  tst r4, #0x4
  beq returnFalse         ; salta si : celda vacia

  ; return si celda != color
  and r5, r4, #0x3
  cmp r5, r3
  bne returnFalse         ; salta si : celda != color

  ; Mientras i < 4
for
  mov r7, #1              ; longLinea := 1
  ldrsb r8, [sp, r6]      ; r8 := deltas_fila[i]
  add sp, sp, #4
  ldrsb r9, [sp, r6]      ; r9 := deltas_columna[i]
  sub sp, sp, #4
  add r1, r1, r8          ; r1 = fila := fila + deltas_fila[i]
  add r2, r2, r9          ; r2 = columna := columna + deltas_columna[i]

  ; Mientras FilaValida & ColumnaValida & CeldaLLena & MismoColor
while
  ; break si fila no valida
  cmp r1, #1
  blo outWhile            ; salta si : fila < 1
  cmp r1, #NUM_FILAS
  bhi outWhile            ; salta si : fila > NUM_FILAS

  ; return si columna no valida
  cmp r2, #1
  blo outWhile            ; salta si : columna < 1
  cmp r2, #NUM_COLUMNAS
  bhi outWhile            ; salta si : columna > NUM_COLUMNAS

  ; r4 := celda
  add r4, r0, r2          ; r4 = r0 + r2 = & cuadricula[0][columna]
  ldrb r4, [r4, r1, LSL#3]; r4 = mem[r4+r1*8] = cuadricula[fila][columna]
  ; return si celda vacia
  tst r4, #0x4
  beq outWhile            ; salta si : celda vacia


```

```
  ; return si celda != color
  and r5, r4, #0x3
  cmp r5, r3
  bne outWhile            ; salta si : celda != color

  add r1, r1, r8          ; fila = fila + delta_fila[i]
  add r2, r2, r9          ; columna = columna + delta_columna[i]
  add r7, r7, #1          ; longLinea++
  ; return si longLinea == 4
  cmp r7, #4
  beq returnTrue          ; salta si : longLinea == 4
  b while

  ; Si !(FilaValida & ColumnaValida & CeldaLLena & MismoColor)
outWhile
  sub r1, r10, r8         ; fila := fila_bak  - delta_fila[i]
  sub r2, r11, r9         ; columna := columna_bak - delta_columna[i]

  ; Mientras FilaValida & ColumnaValida & CeldaLLena & MismoColor
while2
  ; return si fila no valida
  cmp r1, #1
  blo outWhile2           ; salta si : fila < 1
  cmp r1, #NUM_FILAS
  bhi outWhile2           ; salta si : fila > NUM_FILAS

  ; return si columna no valida
  cmp r2, #1
  blo outWhile2           ; salta si : columna < 1
  cmp r2, #NUM_COLUMNAS
  bhi outWhile2           ; salta si : columna > NUM_COLUMNAS

  ; r4 := celda
  add r4, r0, r2          ; r4 = r0 + r2 = & cuadricula[0][columna]
  ldrb r4, [r4, r1, LSL#3]; r4 = mem[r4+r1*8] = cuadricula[fila][columna]
  ; return si celda vacia
  tst r4, #0x4
  beq outWhile2           ; salta si : celda vacia

  ; return si celda != color
  and r5, r4, #0x3
  cmp r5, r3
  bne outWhile2           ; salta si : celda != color

  sub r1, r1, r8          ; fila = fila - delta_fila[i]
  sub r2, r2, r9          ; columna = columna - delta_columna[i]
  add r7, r7, #1          ; longLinea++
  ; return si longLinea == 4
  cmp r7, #4
  beq returnTrue          ; salta si : longLinea == 4
  b while2

  ; Si !(FilaValida & ColumnaValida & CeldaLLena & MismoColor)
outWhile2
  mov r1, r10             ; fila := fila_bak
  mov r2, r11             ; columna := columna_bak
  add r6, r6, #1          ; i++
  ; for i < 4
  cmp r6, #4
  bne for

  ; Si i == 4 || !(FilaValida & ColumnaValida & CeldaLLena & MismoColor)
returnFalse
  mov r0, #0              ; r0 := FALSE
  add sp, sp, #8          ; libera vectores de pila
  LDMIA sp!, {r4-r11, lr} ; Recupera el valor de los registros
  bx lr                   ; return FALSE
  ; Si LongLinea == 4
returnTrue
  mov r0, #1              ; r0 := TRUE
  add sp, sp, #8          ; libera vectores de pila
  LDMIA sp!, {r4-r11, lr} ; Recupera el valor de los registros
  bx lr                   ; return TRUE
```

3.3.2. Optimizaciones aplicadas

Para esta implementación se ha eliminado la recursividad de la función *buscar_alineamiento* y se ha inscrito dentro de *hay_linea*. En la versión original se comprobaba la misma celda para todas las direcciones posibles, algo completamente innecesario. Por eso, antes de comprobar el resto de celdas comprueba que la celda pasada como parámetro es válida para hacer línea. La versión original también cometía el error de seguir comprobando celdas una vez que la longitud de la línea superaba 4. En esta versión, en el momento en el que la longitud de línea encontrada es igual a 4 la función termina la ejecución devolviendo verdad. Además de reducir las iteraciones permite eliminar la variable *"linea"* y simplificar la guarda del bucle for.

El resto de optimizaciones son las mismas que en las anteriores implementaciones: el bucle for realiza siempre una primera ejecución y la guarda de los bucles internos es una secuencia de or lógicos cortocircuitados.

Antes de la implementación en código ensamblador diseñamos una versión en c, para asegurar que el funcionamiento era el esperado:

```
uint8_t conecta4_hay_linea_arm_arm_c(CELDA cuadricula[TAM_FILS][TAM_COLS], uint8_t fila,
    uint8_t columna, uint8_t color)
 {
    int8_t deltas_fila[4] = {0, -1, -1, 1};
    int8_t deltas_columna[4] = {-1, 0, -1, -1};
    unsigned int i = 0;
    uint8_t long_linea = 0;
    uint8_t fila_aux = fila;
    uint8_t columna_aux = columna;

    if (!C4_fila_valida(fila) || !C4_columna_valida(columna) ||
            celda_vacia(cuadricula[fila][columna]) ||
            (celda_color(cuadricula[fila][columna]) != color)) {
        return FALSE;
    }

    // buscar linea en fila, columna y 2 diagonales
    for (i = 0; i < 4; ++i) {
        long_linea = 1;
        fila += deltas_fila[i];
        columna += deltas_columna[i];
        // buscar sentido
        while (!(!C4_fila_valida(fila) || !C4_columna_valida(columna) ||
                celda_vacia(cuadricula[fila][columna]) ||
                (celda_color(cuadricula[fila][columna]) != color))) {
            fila += deltas_fila[i];
            columna += deltas_columna[i];
            long_linea++;
            if (long_linea == 4) return TRUE;
        }

        fila = fila_aux - deltas_fila[i];
        columna = columna_aux - deltas_columna[i];
        // buscar sentido inverso
        while (!(!C4_fila_valida(fila) || !C4_columna_valida(columna) ||
                celda_vacia(cuadricula[fila][columna]) ||
                (celda_color(cuadricula[fila][columna]) != color))) {
            fila -= deltas_fila[i];
            columna -= deltas_columna[i];
            long_linea++;
            if (long_linea == 4) return TRUE;
        }
        fila = fila_aux;
        columna = columna_aux;
    }
    return FALSE;
}
```

Observando el código generado por el compilador de C para esta función se comprobó que la traducción de los bucles while era diferente. En nuestra implementación la guarda del bucle está situada antes del código entre corchetes, lo que implica ejecutar dos saltos en cada iteración. En cambio, el optimizador sitúa la guarda debajo del código y accede a ella por primera vez con un salto incondicional, ahorrando un salto en cada iteración. Sin embargo, más adelante se mostrará que nuestra implementación es un poco más rápida que la

mejor optimización del compilador, pese a ser idénticas en el resto del código. Eso se debe a que en la versión compilada de C se ejecuta después de cada operación aritmética la instrucción **AND Rx,Rx,#0x000000FF**, que trunca el resultado a 8 bits. Por el dominio de valores de las variables del programa esa instrucción no es necesaria, pero el optimizador no puede asumir que no habrá desbordamientos.

4. Testing

Para realizar los diferentes tests y validar el comportamiento de las funciones ha hecho falta modificar la función *C4_verificar_4_en_linea*. En ella se ejecutan todas las versiones de *conecta4_hay_linea*. Si alguna devuelve un resultado diferente, a la variable por referencia *fail* se la asigna verdad, indicando que hay un comportamiento inesperado. El valor devuelto es un and lógico de los 4 resultados, para que en caso de error no se dé por válida la línea, aunque no tiene sentido comprobar el resultado en ese caso.

Para facilitar la localización del error, en la zona de variables estáticas se han declarado las variables *fila_fail*, *col_fail* y *fail*. Si al ejecutar el programa de tests *fail* se activa a 1, finaliza la ejecución y la celda para la que ha ocurrido el error queda visible en *fila_fail* y *col_fail*.

```
int C4_verificar_4_en_linea(CELDA cuadricula[TAM_FILS][TAM_COLS],
                uint8_t fila, uint8_t columna, uint8_t color, int *fail)
{
  uint8_t resultado_c_c     = conecta4_hay_linea_c_c(cuadricula, fila, columna, color);
  uint8_t resultado_c_arm   = conecta4_hay_linea_c_arm(cuadricula, fila, columna, color);
  uint8_t resultado_arm_c   = conecta4_hay_linea_arm_c(cuadricula, fila, columna, color);
  uint8_t resultado_arm_arm = conecta4_hay_linea_arm_arm(cuadricula, fila, columna, color);
  if (fail != 0) *fail = resultado_c_c != resultado_c_arm ||
                         resultado_c_c != resultado_arm_c ||
                         resultado_c_c != resultado_arm_arm;
  return resultado_c_c && resultado_c_arm &&
         resultado_arm_c && resultado_arm_arm;
}
```

```
int test(CELDA cuadricula[TAM_FILS][TAM_COLS], uint8_t f, uint8_t c, const int res)
{
  int fail;
  return res == C4_verificar_4_en_linea(cuadricula, f, c, celda_color(cuadricula[f][c]), &
    fail) && !fail;
}
```

Los tableros empleados en los tests que involucran *C4_verificar_4_en_linea* están declarados en el fichero *tableros.h* y son los siguientes:

Tablero 1 = cuadricula_2

0	C1	C2	C3	C4	C5	C6	C7
F1	6	5	5	6	5	5	6
F2	5	6	5	6	5	6	5
F3	5	5	6	6	6	5	5
F4	6	6	6	5	6	6	6
F5	5	5	6	6	6	5	5
F6	5	6	5	6	5	6	5

Tablero 2 = cuadricula_3

0	C1	C2	C3	C4	C5	C6	C7
F1	5	6	6	6	6	6	5
F2	6	5	0	0	0	5	6
F3	6	0	5	5	5	0	6
F4	6	0	5	5	5	0	6
F5	6	5	0	0	0	5	6
F6	5	6	6	6	6	6	5

Tablero 3 = cuadricula_4

0	C1	C2	C3	C4	C5	C6	C7
F1	6	5	6	5	6	5	6
F2	5	6	5	6	5	6	5
F3	6	5	6	5	6	5	6
F4	5	6	5	6	5	6	5
F5	6	5	6	5	6	5	6
F6	5	6	5	6	5	6	5

4.1. Test 0

Este test fue realizado de manera manual durante el paso 2 y su finalidad era comprobar que la función *C4_comprobar_empate* funcionaba correctamente. Para la comprobación se uso un tablero completo donde el empate se daba en la última fila.

Si en la jugada actual se llena el tablero (no quedan celdas vacias) hay empate. Si no, todavía puede ganar alguno de los jugadores.

4.2. Test 1

Dado el tablero 1, el resultado de ejecutar la función test debe ser FALSE para cualquier posición i, j del tablero.

La finalidad de este test es comprobar que no se avanza en direcciones erroneas detectando falsos ganadores.

4.3. Test 2

Dado el tablero 2, el resultado de ejecutar la función test debe ser TRUE para cualquier posición i, j dentro del array de posiciones que tiene definidas todas las posiciones del tablero que contienen un 5.

La finalidad de este test es detectar que se detectan ganadores para todas las diagonales desde todas sus posiciones de la misma.

4.4. Test 3

Dado el tablero 2, el resultado de ejecutar la función test debe ser TRUE para cualquier posición i, j dentro del array de posiciones que tiene definidas todas las posiciones del tablero que contienen un 6.

La finalidad de este test es detectar ganadores en todas las horizontales y verticales desde todas las posiciones de la misma.

4.5. Test 4

Dado el tablero 2, el resultado de ejecutar la función test debe ser FALSE para cualquier posición i, j dentro del array de posiciones que tiene definidas todas las posiciones del tablero que contienen una celda vacía.

La finalidad de este test es detectar que no se generan ganadores para las celdas vacias.

4.6. Test 5

Dado el tablero 3, el resultado de ejecutar a función test debe ser TRUE para cualquier posición i, j del tablero.

La finalidad de este test es detectar ganadores para cualquier posición del tablero sin salirse de este.

5. Comparación de resultados

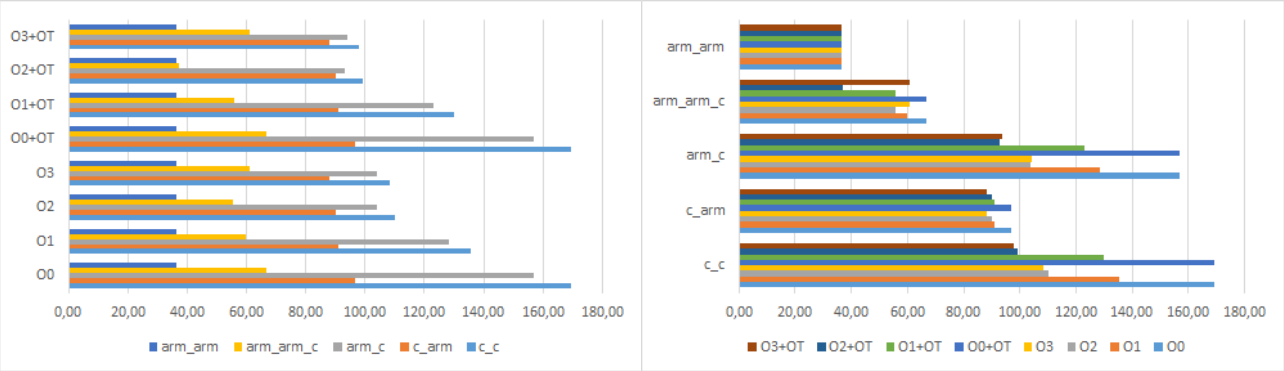
5.1. Tiempo de ejecución

Los tiempos de ejecución han sido medidos para un caso costoso: comprobar tres que en todas las direcciones hay tres casillas en línea, pero no cuatro, y justamente en la última dirección comprobada se encuentra línea. El tablero utilizado es el siguiente:

0	C1	C2	C3	C4	C5	C6	C7
F1	0	5	6	5	6	5	0
F2	0	6	5	5	5	0	0
F3	0	5	5	**0**	0	0	0
F4	0	0	5	0	0	0	0
F5	0	0	0	0	0	0	0
F6	0	0	0	0	0	0	0

Al introducir la siguiente ficha del color 1 en la columna 4 la función *hay_linea* debe encontrar la línea en la última diagonal que se comprueba según la distribución de los deltas. Midiendo los tiempos totales de ejecución en micro segundos, contando tanto el tiempo transcurrido en la función *hay_linea* como en las múltiples llamadas a la función *buscar_alineamiento* se han obtenido los siguientes resultados (en microsegundos):

Función	**-O0**	**-Otime**	**-O1**	**-Otime**	**-O2**	**-Otime**	**-O3**	**-Otime**
hay_linea_c_c	169,50	169,50	135,50	130,08	110,08	99,16	108,25	98,00
hay_linea_c_arm	96,75	96,75	91,16	90,91	90,25	90,25	88,16	88,16
hay_linea_arm_c	156,83	156,83	128,41	123,25	103,91	93,00	104,16	93,91
hay_linea_arm_arm_c	66,75	66,75	59,83	55,91	55,66	37,16	60,91	60,91
hay_linea_arm_arm	36,50	36,50	36,50	36,50	36,50	36,50	36,50	36,50



(a) Comparación por optimización

(b) Comparación por implementación

Figura 3: Comparación de tiempos en µs

Lo primero que se puede observar es que el parámetro de optimización para tiempo mejora el rendimiento en todos los niveles de optimización, salvo para el nivel 0, en el que no se aplica ninguna heurística para facilitar la depuración del código. También se puede ver que el nivel de optimización 2 ha conseguido mejores resultados en tiempo que el nivel 3. Las mejoras en tiempo son más notable cuando la función de c optimizada es *buscar_alineamiento*, ya que es la que más veces se ejecuta.

Si se compara la versión completamente escrita en ensamblador con las demás, se puede ver que había un amplio margen de mejora. Es más de 3 veces más rapida que la implementación original en modo depuración, y el doble de veloz que la siguiente versión más rápida.

5.2. Tamaño del código

En máquinas con poca memoria conviene reducir el tamaño del código. En esta sección se quiere obtener qué función ocupa menos espacio en memoria y razonar si las optimizaciones en tiempo afectan negativamente al tamaño de las funciones. Tras una medición se han obtenido los siguientes resultados:

Función	**Nivel -O**	**Tamaño(bytes)**
buscar_alineamiento_c	-O0	144
	-O0 -Otime	144
	-O1	116
	-O1 -Otime	128
	-O2	**112**
	-O2 -Otime	124
	-O3	**112**
	-O3 -Otime	124
buscar_alineamiento_arm		116
hay_linea_c_c/arm	-O0	320
	-O0 -Otime	320
	-O1	268
	-O1 -Otime	277
	-O2	244
	-O2 -Otime	256
	-O3	232
	-O3 -Otime	244
hay_linea_arm_c		**192**
hay_linea_c_c + buscar_alineamiento_c	-O3	344
hay_linea_c_arm + buscar_alineamiento_arm	-O3	348
hay_linea_arm_c + buscar_alineamiento_c	-O3	**304**
hay_linea_arm_arm_c	-O3	352
hay_linea_arm_arm		348

En primer lugar se han comparado las funciones recursivas *buscar_alineamiento*. El resultado obtenido muestra que las optimizaciones realizadas por el compilador con la opción -O2 y -O3 son las que más reducen el tamaño de la función. En el caso de las funciones *hay_linea* triunfa nuestra implementación. En cuanto a la mejora según el nivel de optimización se puede concluir que cuanto mayor es el nivel, menor es el tamaño del código, y que cuando el flag -Otime está activo siempre aumenta el tamaño en pos de mejorar el tiempo de ejecución.

Por último se ha comparado el tamaño total de las funciones, medidas en -O3 por ser el nivel de optimización que más reducía el código. Cuando una función llama a la otra los bytes empleados se reducen en proporción a la mejora individual de cada una. Sin embargo, cuando se elimina la recursividad y se inscribe el código de *buscar_alineamiento* dentro de *hay_linea*, como en el caso de *hay_linea_arm_arm*, aumenta el tamaño de la función. Esto se debe principalmente a que originalmente había dos llamadas a la función recursiva, lo que implica integrar la función dos veces. Una versión que simplemente eliminara la recursividad habría reducido con creces el tamaño del código, pero no obtendría unos resultados tan bajos en tiempo.

6. Problemas y soluciones

Al realizar las mediciones de tiempo y tamaño había algunas funciones que el compilador las ponía inline por lo que se tuvo que añadir la orden *__attribute__ ((noinline)* para que el keil pudiera detectar bien los tiempos y a su vez disponer correctamente de la traducción a ensamblador para medir el espacio que ocupaba el código.

También surgió otro problema con la pila al implementar una de las funciones en ensamblador que solucionamos forzando el alineamiento a 8 bytes mediante la directiva *PRESERVE8{TRUE}*.

Los tests fueron de ayuda para depurar lo siguientes errores:

- Para negar el valor de *deltas_fila[i]* y *deltas_columna[i]* se utilizó la función SBC por confundir su funcionamiento con el de RSB.
- Las funciones no consideraba válidas las celdas de la última columna por comparar con *NUM_FILAS* en lugar de *NUM_COLUMNAS*.
- La recuperación del valor de los parámetros no se tuvo en cuenta hasta encontrar un error al invocar desde *hay_linea_arm_c* a la función *buscar_alineamiento_c*, cuando el valor del parámetro color contenido en r3 cambiaba tras la invocación.
- Surgió un error al reutilizar la condición de la función recursiva, que utilizaba r4 para contener el valor de la celda, en la última versión de *hay_linea*, que utilizaba r6.

7. Conclusiones

Tras haber completado la práctica, nuestras conclusiones se resumen principalmente en que el uso adecuado de las opciones del compilador puede ayudar considerablemente a aumentar el rendimiento de un programa, y que cuando programamos en ensamblador siempre podemos tener en cuenta factores que el compilador desconoce para optimizar todavía más el código original.

Entre los conocimientos adquiridos no consideramos el uso de técnicas de optimización tales como eliminar la recursividad, reformular las condiciones y otras técnicas utilizadas en la práctica, debido a que ya se nos dieron a conocer en asignaturas pasadas. Sin embargo, sí que ha resultado un buen ejercicio para demostrar su potencial.

8. Dedicación

En la siguiente tabla se muestra la distribución de las horas de trabajo a lo largo de los pasos señalados en el enunciado de la práctica. En general el trabajo ha sido realizado entre los dos integrantes a la vez para evitar el mayor número de fallos posible en la metodología y las mediciones, así como detectar al instante errores de código.

Tarea	**Fernando**	**Héctor**	**Comentario**
Paso 1	4	3	Estudiar la documentación
Paso 2.1	3	3	Estudiar el juego y función comprobar empate
Paso 2.2	2	2	Mapa de memoria y marco de activación
Paso 3	1	1	Analizar el rendimiento
Paso 4	2	2	Función conecta4_buscar_alineamiento_arm
Paso 5	2	2	Función conecta4_hay_linea_arm_c
Paso 6	3	3	Función conecta4_hay_linea_arm_arm
Paso 7	3.5	3.5	Verificación automática
Paso 8	2	2	Medidas de rendimiento
Paso 9	1	1	Optimizaciones del compilador
Memoria	11.5	10	
Total	**35**	**32.5**	