# GUIÓN 1: DESARROLLO DE CÓDIGO PARA EL PROCESADOR ARM

## INTRODUCCIÓN

La asignatura Proyecto Hardware tiene por objetivo desarrollar un proyecto que permita evaluar el uso óptimo y eficiente del procesador y sus periféricos. Para gestionar mejor el proyecto está dividido en varias etapas con entregas parciales.

Este guion corresponde al trabajo de las tres primeras sesiones de laboratorio, en las que vamos a optimizar el rendimiento de un juego acelerando las funciones computacionalmente más costosas. A partir de un código facilitado en C, se desarrollará código en ensamblador ARM para las funciones más críticas y se ejecutará sobre un emulador de un System-on-Chip ARM con el entorno de desarrollo Keil µVision IDE. También compararemos el rendimiento obtenido, con el que consigue el compilador al activar sus opciones de optimización. Finalmente, en todas las entregas de la asignatura, os debéis asegurar de que vuestro código funciona correctamente, para ello, en esta primera entrega desarrollaréis una función que verifique que todas las versiones del código den un resultado equivalente.

## ÍNDICE

## OBJETIVOS

- Interactuar con un microcrontrolador y ser capaces de **ejecutar** y **depurar**.
- Profundizar en la interacción C / Ensamblador.
- Ser capaces de depurar el **código ensamblador** que genera un compilador a partir de un lenguaje en alto nivel.
- Saber **depurar** código siguiendo el estado arquitectónico de la máquina: contenido de los registros y de la memoria.
- Conocer la estructura segmentada en tres etapas del procesador ARM 7, uno de los más utilizados actualmente en sistemas empotrados.
- Familiarizarse con el entorno Keil µVision sobre Windows, con la generación cruzada de código para ARM y con su depuración.
- Aprender a analizar el rendimiento y la estructura de un programa.
- Desarrollar código en ensamblador ARM: adecuado para optimizar el rendimiento.
- **Optimizar** código: tanto en ensamblador ARM, como utilizando las opciones de optimización del compilador.
- Entender la finalidad y el funcionamiento de las ***Aplication Binary Interface***, ABI, en este caso el estándar **ATPCS** (*ARM-Thumb Application Procedure Call Standard*), y combinar de manera eficiente código en ensamblador con código en C.
- Comprobar automáticamente que varias implementaciones de una función mantienen la misma funcionalidad.

## CONOCIMIENTOS PREVIOS NECESARIOS

En esta asignatura cada estudiante necesitará aplicar contenidos adquiridos en asignaturas previas, en particular, Arquitectura y Organización de Computadores I y II.

## ENTORNO DE TRABAJO

Esta práctica se realiza con **Keil µVision**, el mismo entorno que ya se utilizó en primero para AOC1. Trabajaremos sobre el microcontrolador LPC2105 (el mismo que se usaba en primero).

El entorno µVision es capaz de editar, depurar, compilar código en C y ensamblador, además de analizar el rendimiento de los binarios generados. En prácticas sucesivas veremos como también es capaz de emular diferentes periféricos y entrada salida del microcontrolador.

Se utilizará la versión 5. En Moodle de la asignatura tenéis información del proceso de instalación y fuentes. También podéis registraros e instalar el software desde la web de Keil.

## MATERIAL ADICIONAL

En el curso Moodle de la asignatura puede encontrarse el siguiente material para prácticas:

- Manuales de Keil µVision
- Documentación del sistema emulado microprocesador LPC2105
- Manuales de la arquitectura de referencia:
    - Breve resumen del repertorio de instrucciones ARM.
    - Manual de la arquitectura ARM.
    - Información sobre el ABI de ARM: ATPCS
    - Documento con información sobre variables en C
- Directrices para redactar una memoria técnica, imprescindible para redactar la memoria de la práctica. Es donde se define la estructura de la memoria a entregar.
- Wikis con información técnica adicional

Y para la realización de la práctica 1 además: proyecto para Keil µVision con los códigos fuentes del juego, este documento, las diapositivas de la presentación y una wiki que resuelve algunas de las dudas más habituales.

## ESTRUCTURA DE LA PRÁCTICA

La práctica completa se debe realizar en **3 sesiones**. En la siguiente sesión tendrá lugar la entrega presencial del trabajo y una semana después se entregará la memoria a través de Moodle.

Antes de la **primera sesión** es imprescindible haberse leído este guion y conocer la documentación suministrada. Para entrar en la **segunda sesión** de prácticas es necesario traer el código fuente de los pasos 4 y 5 lo más avanzados posible y se presentarán los pasos previos. La **tercera sesión** se debería dedicar a realizar las métricas de los apartados 7 y 8, por lo que se deberá mostrar antes de empezar el correcto funcionamiento de hasta el punto 6 incluido.

## EL JUEGO: CONECTA K EN LINEA

Seguro que conoces el Conecta 4. En este caso vamos a trabajar con el Conecta K, un juego de la misma familia, que se juega en un tablero plano (sin gravedad) y donde se pueden colocar las fichas en cualquiera de las posiciones vacías.

Son toda una familia de juegos, donde cada uno viene definido por $(m, n, k, p, q)$, donde $m$ x $n$ es el tamaño del tablero, cada jugador en su turno coloca $p$ fichas salvo la primera jugada en la que se colocan $q$. Gana el juego quien alinee $k$ fichas del mismo color.

Uno de los juegos más famoso es el `Conecta 6` o `Conecta (∞,∞, 6, 2, 1)`, se juega en un tablero infinito, se colocan dos fichas en cada turno salvo el primero que solo pone una. El objetivo es alinear 6 fichas.

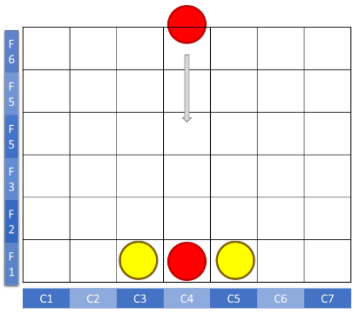


Figura 1 Ejemplo de conecta 4

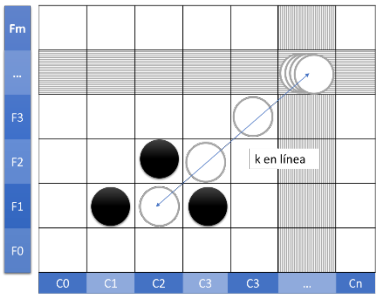


Figura 2 Esquema de tablero de conecta( m,n,k,p, q)

## ¿CÓMO FUNCIONA EL CÓDIGO FACILITADO?

El código suministrado es el esqueleto para cualquier juego de la familia Conecta K. El código que realicéis debe ser capaz de jugar a cualquier juego de la familia. Las constantes `m, n, k, p y q` están definidas en el fichero `config_conecta_K.h`. Inicialmente estamos jugando al `Conecta K (7, 7, 4, 2, 1)`.

Para esta primera práctica no utilizaremos el sistema de entrada/salida. Para facilitar las pruebas y depuración del juego emulamos la iteración del usuario escribiendo y leyendo directamente de memoria. Keil permite visualizar y modificar regiones de memoria durante la depuración. La función inicial del juego define las variables **`entrada`** y **`salida`** alineadas a 8 bytes para su correcta visualización en memoria.

La entrada se realiza mediante el vector de `entrada` y sus funciones de acceso definidas en `entrada.h`. La información del tablero está guardada en una estructura de datos denominada `cuadricula`, con funciones definidas en `tablero.h`, consistente en una representación de una matriz dispersa de `celdas`. Visualizar el tablero para poder ir viendo el desarrollo del juego sobre dicha estructura no es fácil. Por ello tras cada jugada vamos a "pintar" una parte del tablero en pantalla. Utilizaremos la matriz `salida` que representa una pequeña pantalla en la memoria. Tras cada jugada dibujaremos sobre esta pantalla el resultado de un trozo inicial del tablero de 7x7 (más indicativos de fila y columna).

Para jugar hay que buscar la dirección del array *entrada* (`0x40000000` en la figura, pero el valor exacto dependerá del código y del compilador) y de la matriz *salida* (`0x40000008` en la figura). En el código inicial aparecen en direcciones consecutivas para facilitar la interacción. Debéis añadir un *breakpoint* en el bucle *while* que espera una entrada nueva y modificar manualmente en memoria el contenido de ***`entrada`***[0], un 1 indicará que hay una nueva jugada que debe procesarse, y de las tres siguientes posiciones de `entrada`, indicando la fila, la columna (ambas empiezan en la 1) y el color (1 o 2) de la ficha elegida.

Una vez validada e introducida la entrada, se realizará la jugada actualizando el tablero y se visualizará el resultado. Posteriormente comprobaremos si hay K fichas alineadas y verificaremos el correcto funcionamiento de las diferentes versiones.

Para comprobar el funcionamiento correcto nos interesa verificar el resultado con una serie de jugadas que prueben casos concretos con resultado conocido buscando la mayor cobertura del código y de los casos frontera.

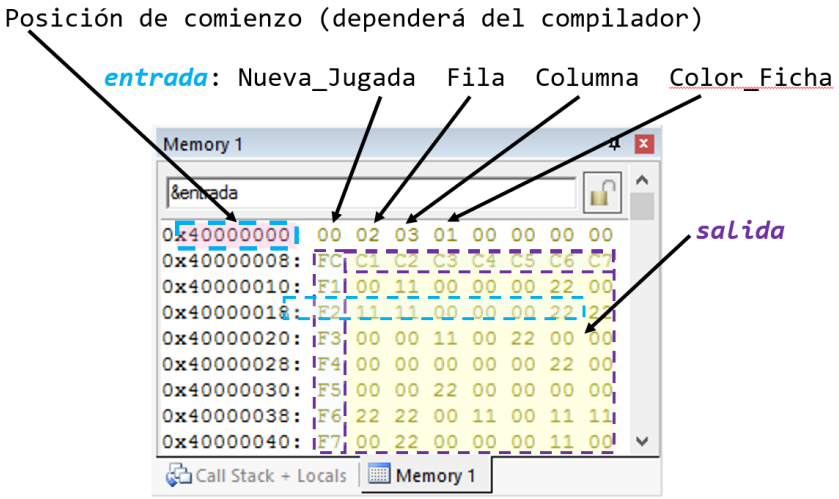


**Figura 3 visualización de la *entrada* y del *salida* de la pantalla como debe verse en memoria. Nota: en la entrada los jugadores se indican con 1 y 2, pero, para mejorar la visualización, en pantalla debe mostrarse un "11" y un "22".**

Se pueden verificar manualmente, introduciendo jugadas una a una, pero es conveniente para acelerar el proceso partir de una partida a medias con un conjunto de fichas ya colocadas. Una forma de hacerlo en mediante un vector de jugadas (fila, columna, color y resultado esperado), y con ello testear el correcto funcionamiento.

Otra forma de comprobar el funcionamiento es precargar un tablero en un estado a mitad de partida. Os proporcionamos una partida inicial en el fichero `tablero_test.h`, deberéis hacer una función que precargue este tablero antes de empezar a jugar. Os vendrá bien para las pruebas iniciales y para poder mostrar el día de la corrección todos los grupos las mismas jugadas.

## TAREAS A REALIZAR

En esta primera parte del proyecto nos centraremos en la implementación eficiente de las funciones más críticas del juego. Para ello primero buscamos entender el funcionamiento del sistema y el comportamiento de las funciones resaltadas. Posteriormente evaluaremos el rendimiento y estudiaremos las opciones de optimización del compilador para mejorar las prestaciones del código generado.

Por tanto, tenéis que:

**Paso 1: Estudiar la documentación**.

**Paso 2: Estudiar y depurar el código inicial del juego.**

Estudiar el código inicial en C y la especificación de las funciones `del juego.` En el código inicial en C debéis analizar y entender todo el código y su modularidad, así como la estructura de datos dispersa para almacenar el tablero. En especial nos interesan las funciones `conecta_K_jugar`, **`conecta_K_hay_linea_c_c`**, y **`conecta_K_buscar_alineamiento_c`.** Debéis comprobar el correcto funcionamiento.

a) Realizad las funciones **`conecta_K_test_cargar_tablero`** que nos permitirá empezar a depurar con una partida ya inicializada y **`conecta_K_visualizar_tablero`** que visualiza en memoria un trozo del tablero para facilitar la depuración. Realizar ambas

funciones nos ayudara a entender las estructuras de datos empleadas y la forma correcta de acceso a ellas.

b) Aprended a jugar interaccionando con el juego a través de la memoria.

c) Debéis monitorizar la ejecución de las **funciones en C, verificando** sobre el tablero en memoria **la correcta ejecución**.

d) Observad el código en ensamblador generado por el compilador y depurarlo paso a paso, prestar atención a los cambios en memoria y en los registros del procesador.

e) Dibujad el mapa de memoria (código, variables globales, pila, etc).

**f)** Dibujad el marco de activación en pila y el paso de parámetros de las llamadas a funciones que está empleando el compilador. **En la memoria comparar este marco con el que conocéis de AOC1 y con el que os hemos dado en las transparencias.**

El mapa de memoria y marco de activación debe enseñarse en las entregas e incluirse en la memoria.

**Paso 3: Analizar el rendimiento.** Hacer uso del analizador de rendimiento (***profiling***) del Keil para analizar el árbol de ejecución y los tiempos de ejecución de las diferentes funciones del juego. Determinar las funciones principales del juego y las de mayor peso de cómputo. En la memoria deberéis reflejar los resultados obtenidos.

**Paso 4: Implementar la función `conecta_K_buscar_alineamiento_arm` en ensamblador ARM.** Esta función debe ser equivalente a la versión en C, incluyendo las funciones de acceso a la estructura de datos en ensamblador. Debéis tratar de optimizar el código. Crear una nueva versión en C llamada `conecta_K_hay_linea_c_arm`, que invoque a `conecta_K_buscar_alineamiento_arm`.

**Paso 5: Implementar la función `conecta_K_hay_linea_arm_c` en ensamblador ARM.** Debéis tratar de optimizar el código. La función invocará a **`conecta_K_buscar_alineamiento_c`.**

El código ARM debe tener la misma estructura que el código C: cada función, cada variable o cada condición que exista en el código C debe poder identificarse con facilidad en la versión de ensamblador. Para ello se deberán incluir los comentarios oportunos.

**Paso 6: Implementar la función `conecta_K_hay_linea_arm_arm`.** Diseña una función en ARM que sustituya a las iniciales de C devolviendo la misma salida. Puedes aplicar cualquier optimización, e incluso eliminar la recursividad.

IMPORTANTE: **ES OBLIGATORIO CUMPLIR EL ATPCS EN LAS LLAMADAS A FUNCIÓN**. Cuando hagáis la llamada en ensamblador debe ser una llamada convencional a una función pasando los parámetros a través de los registros correspondientes o la pila según el estándar **ATPCS**. No sirve hacer un salto sin pasar parámetros, ni pasarlos de una forma distinta al estándar.

El marco de pila (o bloque de activación) de las distintas combinaciones debe ser idéntico al creado por el compilador para que la comparación sea justa. **La descripción del marco de pila utilizado deberá aparecer convenientemente explicado en la memoria.**

En todos los casos se debe garantizar que una función no altere el contenido de los registros privados de las otras funciones.

**Paso 7: Verificación automática y comparación de resultados.** Los procesos de verificación y optimización son una parte fundamental del desarrollo del software y deben tenerse en cuenta desde el principio del tiempo de vida del software.

Tenemos diversas implementaciones de **`conecta_K_hay_linea`**(C-C, C-ARM, ARM-C y ARM-ARM). Como durante la fase de desarrollo del código en ARM es muy fácil que se cometa algún error **no** fácilmente **detectable al depurar** a mano. Vuestro código debe **verificar que las distintas combinaciones generan la misma salida.** Este proceso lo tendréis que realizar múltiples veces, por lo que **se debe automatizar**.

Para ello, la función **`conecta_K_verificar_K_en_linea`** debe realizar esta tarea invocando a las diferentes versiones en cada jugada. Dentro de esta función se llamará a las diferentes versiones comprobando que el resultado coincide con la solución esperada.

Consejo: antes de verificar **`conecta_K_hay_linea, os aconsejamos verificar bien`** `conecta_K_buscar_alineamiento_arm` siguiendo un enfoque botton_up (verifica primero los módulos más sencillos) y comparándola con `conecta_K_buscar_alineamiento_c`.

Diseñad un banco de pruebas que compruebe los casos que consideréis relevantes. Incluid una descripción de estos casos en la memoria. Podéis generar nuevos tableros, o vectores de jugadas modificando el mecanismo de la entrada del juego.

**Paso 8: Medidas de rendimiento.** Una vez comprobado que vuestro código funciona bien y que el entorno mide bien los tiempos, vamos a medir los tiempos de ejecución sobre el procesador.

Nos interesa medir las funciones críticas (no en todo el programa completo). Debéis comparar el **tamaño del código** en **bytes** de cada versión de las funciones realizadas (C original, ARM, etc). También queremos **evaluar el tiempo de ejecución** de cada una de estas configuraciones. Calcular las métricas de las combinaciones, comparar y **razonar los resultados obtenidos.**

Los valores del tamaño de código y los tiempos de ejecución debéis presentarlos en la entrega de la práctica **el día de la corrección en el laboratorio**.

**Paso 9: Optimizaciones del compilador.** Por defecto el compilador de C genera un código seguro y fácilmente depurable (*debug*[1]) facilitando la ejecución paso a paso en alto nivel. Esta primera versión es buena para depurar, pero es muy ineficiente. Los compiladores disponen de diversas opciones de compilación o *flags* que permiten configurar la generación de código. En concreto hay un conjunto de opciones que permiten aplicar heurísticas de optimización de código buscando mejorar la velocidad o el tamaño. Estas configuraciones se suelen especificar con los *flags* `-O0` (por defecto, sin optimizar), `-O1`, `-O2` (código en producción) y `-O3`.

[1] Te sorprendería saber la cantidad de software comercial generado sin optimizar y en modo debug.

**Estudiar el impacto en el rendimiento del código C de los niveles de optimización** (`-O0, -O1, -O2, -O3, -optimize for time`) y compararlo con las otras versiones en rendimiento, tamaño y número de instrucciones ejecutadas. Los cambios en el nivel de optimización sólo deberían afectar al rendimiento del código de alto nivel escrito en C y la funcionalidad no debería cambiar. Sin embargo, a veces aparecen errores. Estos errores no son culpa del compilador, son errores que ya estaban en el código pero que no eran visibles. Los errores más típicos son no seguir correctamente el ATPCS, o no asignar **`volatile`** a una variable que cambia externamente (esto os pasará más adelante). La opción `-Otime`, `Optimize for time` en los menús de Keil, permite realizar optimizaciones adicionales que pueden incrementar el tamaño de los binarios como loop unrolling o inlining. Para ver estos efectos debéis por favor activar "`Optimize for time`" al activar el nivel de optimización O3.

Todos estos resultados deben estar disponibles **el día de la corrección** y quedar claramente reflejados y comentados **en la memoria**.

Nota: no ganar al compilador en `-O3` no es un fracaso. Los compiladores son muy buenos optimizando código. El objetivo no es tanto que ganéis al compilador, como que conozcáis estos flags de optimización y los uséis en el futuro.

## PUNTOS DE VERIFICACIÓN EN CADA ENTREGA

### ENTREGA 1: INICIO DE LA SESION 2 (SEMANA 3 DEL CALENDARIO EINA)

Antes del inicio de la segunda sesión debéis haber realizado correctamente y verificado los pasos 1, 2 y 3. Así mismo estaréis trabajando ya en los pasos 4 y 5 realizando el código en ARM de las funciones básicas, aunque estén sin acabar de depurar u optimizar. Se verificará:

- El funcionamiento del código de visualización en pantalla (memoria) para el tablero inicial localizado en `tablero_test.h`.
- El rendimiento de la ejecución, en particular sobre el tablero inicial de la jugada del jugador 2 colocando ficha en la fila 4, columna 4, indicando las funciones más costosas.
- El código en ARM de las funciones **`conecta_K_buscar_alineamiento_arm`** y **`conecta_K_hay_linea_arm_c`**.
- Que entendéis el interfaz binario **ATPCS** utilizado por el compilador y vuestro código ARM (uso de registros, paso de parámetros y resultado), se mostrará al profesor el marco de pila utilizado (en papel o digital).

### ENTREGA 2: INICIO DE LA SESION 4

Se deben mostrar

- Los tiempos de ejecución de las distintas versiones de las funciones y niveles de optimización del compilador sobre el tablero inicial de la jugada del jugador 2 colocando ficha en la fila 4, columna 4. Formato tabla en papel o digital.
- Las tres principales optimizaciones implementadas en la versión ARM_ARM.
- Diagrama y justificación de la verificación automática implementada.

## APARTADO OPCIONAL 1:

Hemos estudiado el impacto en el rendimiento del código C de las opciones compilación con optimización (-O1, -O2…). La gran ventaja de que sea el compilador el que aplique las optimizaciones es que nos permite mantener claridad y portabilidad en el código fuente, mientras obtenemos rendimiento en el ejecutable. Si hay algún cambio, lo único que deberemos hacer es recompilar para sacar de nuevo partido.

Conforme el compilador aplica las heurísticas el código se hace menos claro. Mira el manual y el código en ensamblador generado por el compilador. Explica qué técnicas ha aplicado en cada versión y como las ha empleado.

Busca optimizar tus funciones ARM aplicando las mismas técnicas. ¿Eres capaz de mejorar el rendimiento obtenido por el compilador?

En algunos casos, nosotros tenemos información sobre el algoritmo o la implementación que el compilador desconoce o debe ser conservador. ¿Eres capaz de optimizar las funciones en C para ayudar al compilador a generar "mejor" código?

## EVALUACIÓN DE LA PRÁCTICA

La práctica se presentará (entrega 2) en la sesión de prácticas entre el 3 y 6 de octubre y se entregará el código. La memoria habrá que presentarla antes del 11 de octubre. Se entregará en Moodle.

En la presentación se entregarán los códigos fuentes y se comprobará que funciona, que al compilar no aparecen *warnings* y que el código es correcto, así como que el trabajo presentado cumple los requisitos de este documento.

**Las fechas y horarios definitivos se publicarán en la página web de la asignatura (*moodle*).**

## ANEXO 1: RECOMENDACIONES PARA LA REALIZACIÓN DE LA MEMORIA

La memoria de la práctica tiene diseño libre, la estructura de la memoria se define en el documento **Ayuda_elaboracion_memoria_tecnica.pdf** disponible en la página web de la asignatura (moodle), y es obligatorio que incluya los siguientes contenidos:

1. Resumen ejecutivo (una cara como máximo). El resumen ejecutivo es un documento independiente del resto de la memoria que describe brevemente qué habéis hecho, por qué lo habéis hecho, qué resultados obtenéis y cuáles son vuestras conclusiones.
2. Código fuente comentado. Además, cada función debe incluir una cabecera en la que se explique cómo funciona, qué parámetros recibe y dónde los recibe y para qué usa cada registro (por ejemplo, en el registro 4 se guarda el puntero a la primera matriz…).
3. Mapa de memoria
4. Descripción de las optimizaciones realizadas al código ensamblador.
5. Análisis de rendimiento y comparativa de las distintas versiones y opciones de compilación (tiempo de ejecución, tamaño de código, etc.)
6. Descripción de los problemas encontrados en la realización de la práctica y sus soluciones.

7. Conclusiones. Este apartado os suele costar. El objetivo es remarcar aquellos mensajes que queráis que se queden en la cabeza del lector. Pensad que es lo último que va a leer. Algunos temas de los que podéis hablar son: ¿para qué ha servido vuestro trabajo?, ¿qué habéis conseguido?, ¿qué opináis de los resultados?, ¿y de la práctica que os hemos pedido?

Se valorará que el texto sea **claro y conciso**. Cuánto más fácil sea entender el funcionamiento del código y vuestro trabajo, mejor. Una memoria razonable no debe superar las 10 páginas (se puede ampliar con código y figuras)

## ANEXO 2: ENTREGA DEL CÓDIGO Y DE LA MEMORIA

La entrega de la memoria será a través de la página web de la asignatura (*moodle* en http://add.unizar.es). Debéis enviar un fichero comprimido en formato ZIP con los siguientes documentos:

1. Memoria en formato PDF.
2. Proyecto y código fuente de los apartados A y B.

Se debe mandar un único fichero por pareja. El fichero se nombrará de la siguiente manera:

p1_NIP-Apellidos_Estudiante1_NIP-Apellidos_Estudiante2.zip

Por ejemplo: p1_345456-Gracia_Esteban_45632-Arribas_Murillo.zip

Los ficheros independientes deben también identificar los nombres de la pareja, por ejemplo: Memoria_p1_ NIP-Apellidos_Estudiante1_NIP-Apellidos_Estudiante2.pdf para el archivo de la memoria.

## ANEXO 3: ALGUNOS CONSEJOS DE PROGRAMACIÓN EFICIENTE PARA OPTIMIZAR EL CÓDIGO ENSAMBLADOR

A la hora de evaluar este trabajo se valorará que cada estudiante haya sido capaz de optimizar el código ARM. **Cuanto más rápido** y pequeño sea el código (y menos instrucciones se ejecuten) **mejor será la valoración de la práctica**. También se valorará que se reduzcan los accesos a memoria.

Algunas ideas que se pueden aplicar son:

- No comencéis a escribir el código en ensamblador hasta tener claro cómo va a funcionar. Si diseñáis bien el código a priori, el número de instrucciones será mucho menor que si lo vais escribiendo sobre la marcha.
- Optimizar el uso de los registros. Las instrucciones del ARM trabajan principalmente con registros. Para operar con un dato de memoria debemos cargarlo en un registro, operar y, por último, y sólo si es necesario, volverlo a guardar en memoria. Mantener en los registros algunas variables que se están utilizando frecuentemente permite ahorrar varias instrucciones de lectura y escritura en memoria. Cuando comencéis a pasar del código C original a ensamblador debéis decidir qué variables se van a guardar en registros, tratando de minimizar las transferencias de datos con memoria.
- Utilizar instrucciones de transferencia de datos múltiples como LDMIA, STMIA, PUSH o POP que permiten que una única instrucción mueva varios datos entre la memoria y los registros.

- Utilizar instrucciones con ejecución condicional, también llamadas instrucciones predicadas. En el repertorio ARM gran parte de las instrucciones pueden predicarse. Por ejemplo, el siguiente código:

```
if (a == 2) { b++ }
else { b = b - 2 }
```

Con instrucciones predicadas sería:

```
CMP   r0, #2            #compara con 2
ADDEQ r1,r1,#1          #suma si r0 es 2
SUBNE r1,r1,#2          #resta si r0 no es 2
```

Mientras que sin predicados sería:

```
CMP   r0, #2      #compara con 2
BNE   resta       # si r0 no es 2 saltamos a la resta
ADD   r1,r1,#1    #suma 1
B     cont        #continuamos la ejecución sin restar
Resta: SUBNE      r1,r1,#2    #resta 2
Cont:
```

Hay otros ejemplos útiles en las transparencias de la práctica.

- Utilizar instrucciones que realicen más de una operación. Por ejemplo la instrucción MLA r2,r3,r4,r5 realiza la siguiente operación: r2 = r3*r4 + r5.
- Utilizar las opciones de desplazamiento para multiplicar. Las operaciones de multiplicación son más lentas (introducen varios ciclos de retardo). Para multiplicar/dividir por una potencia de dos basta con realizar un desplazamiento que además puede ir integrado en otra instrucción. Por ejemplo:
    - A= B + 2*C puede hacerse sencillamente con ADD R1, R2, R3, lsl #1
    - A= B + 10*C puede hacerse sencillamente con ADD R1, R2, R3, lsl #3 y ADD R1, R1, R3, lsl #1 (A = B + 8*C + 2*C)

Sacar partido de los modos de direccionamiento registro base + offset en los cálculos de la dirección en las instrucciones load/store. Por ejemplo, para acceder a A[4] podemos hacer LDR R1, [R2, #4] (si es un array de elementos de 8 bits) o LDR R1, [R2, #16] (si es un array de elementos de 32 bits). Además, en ARM se puede desplazar uno de los operandos de la dirección para hacer multiplicaciones de potencias de 2.

NOTA: como el código a desarrollar es pequeño es probable que alguna de estas optimizaciones no sea aplicable a vuestro código, pero muchas sí que lo serán y debéis utilizarlas.