# PRACTICA 1

# PROYECTO HADWARE

Diego Santolaya Martínez (727209)

Abel Chils trabanco (718997)

**ÍNDICE:**

# 1. Resumen

En la realización de este trabajo se ha analizado el código de un programa que juega al reversi, más concretamente de dos funciones de este programa, que son patrón volteo y ficha válida.
Se han codificado ambas funciones en ARM y la segunda también en THUMB, siguiendo las directrices del enunciado, tras lo cual se verificó su correcto funcionamiento.
La depuración ha sido el proceso más largo y más importante ya que se ha invertido mucho tiempo optimizar el código y comprobar su correcto funcionamiento. Para medir los tiempos que tardaba cada combinación de los distintos lenguajes se ha utilizado un bucle que repetía la función 1.000.000 de veces y un cronometro debido a que al no haber sistema operativo no se puede acceder a bibliotecas que proporcionen el tiempo del sistema.
Los distintos tipos de optimización en compilación se han probado para medir el número de instrucciones totales de cada función y el número de instrucciones ejecutadas al llamar a patrón volteo con los datos especificados. Posteriormente se analiza la repercusión de estas optimizaciones en el ensamblador y se realizan unos programas de prueba con varios casos para hacer las pruebas de las funciones automáticamente.

# 2. Introducción

El trabajo a realizar se compone de 3 partes:

La primera consiste en pasar las dos funciones ficha válida y patrón volteo a código ensamblador, ambas en ARM y sólo la primera en THUMB y comprobar su correcto funcionamiento.
La segunda trata de medir los tiempos de ejecución de las distintas combinaciones (C-C, C-ARM, C-THUMB, ARM-C, ARM-ARM, ARM-THUMB) y contar el número de instrucciones totales y de ejecución con los distintos grados de optimización de la combinación C-C.
Por último, los apartados voluntarios consisten en realizar unos programas que comprueben automáticamente la corrección de las funciones y analizar las repercusiones de los distintos grados de optimización en el ensamblador, así como realizar el mismo análisis que se realiza en el apartado anterior sobre la combinación C-C, pero esta vez sobre la combinación C-ARM.
En esta memoria se intenta explicar detalladamente los pasos seguidos, las decisiones tomadas, los resultados obtenidos y las conclusiones extraídas de este proyecto.

# 3. Objetivos

Se ha tratado de representar fielmente las funciones proporcionadas en C al pasarlo a ensamblador, realizando cada asignación, cada comparación y cada llamada que se hacen en las funciones originales. Uno de los objetivos fundamentales que se ha planteado ha sido la de la optimización del código, tratando de aplicar todas las mejoras posibles para reducir el código al máximo, para ello se ha hecho uso de la ejecución condicional, así como evitar operaciones costosas como pueden ser las multiplicaciones y los accesos a memoria.

Con el fin de comprender como afectan los distintos grados de optimización y las distintas combinaciones (C, ARM o THUMB) a la ejecución del código se han contabilizado el número de instrucciones y se ha medido el tiempo de ejecución.

# 4. Metodología

Tras unas horas de analizar el código, se prosiguió a realizar un esquema de lo que debía de hacer cada función y cómo se podría hacer en ensamblador de la manera más eficiente posible, tras lo que se consultó en los manuales de la arquitectura las restricciones de cada instrucción, así como la sintaxis y algún ejemplo para asegurar la corrección del código.

Tras la realización de la función de ficha válida y de la corrección de esta con un pequeño programa de prueba (ejecutándolo en modo depuración), se procedió a implementar la función patrón volteo, más compleja por las distintas llamadas a funciones, con lo que conlleva el siguiente tratamiento de la pila y del contexto.

Para asegurar que durante todo el proceso todas las combinaciones se comportaban igual se decidió crear en el programa de pruebas dos punteros a funciones, los cuales se llaman desde las funciones patrón volteo y patrón volteo ARM, permitiendo en tiempo de ejecución ir generando todas las combinaciones.

Para medir el número de instrucciones de cada grado de optimización se ha observado detenidamente el desensamblado, se han contado a mano instrucciones y restando posiciones de memora para cerciorarse de su valor y se ha utilizado el trozo bucle proporcionado en el enunciado para contar el número de instrucciones en ejecución.

Guardando cada desensamblado y comparándolos unos con otros, además de mirar la tabla de optimizaciones aplicadas para cada grado (O1, O2, O3 y s), se comprueba en el código las mejoras aplicadas y como a veces estás pueden llevarnos a errores (ya que en O2, O3 a veces el compilador es muy agresivo y omite partes del código que no se

esperan que se omitan), como ya se particulariza más adelante en los apartados optativos.

# 5. Diseño del código

## 5.1 Función ficha válida ARM

Para el diseño de esta función se tradujo cada sentencia del código de C a ARM.

La primera sentencia que se ejecuta en C corresponde con una condición: if ((f < DIM) && (0 <= f) && (c < DIM) && (0 <= c) && (tablero[f][c] != CASILLA_VACIA),

Para traducirla a ensamblador se dividió en dos bloques, el primero para comprobar la condición (f < DIM) && (0 <= f) && (c < DIM) && (0 <= c), y el segundo para comprobar si tablero[f][c] != CASILLA_VACIA.

El primer bloque se logró implementar con 3 instrucciones, gracias a la ejecución condicional y el hecho de que, si se interpreta un dato de tipo CHAR negativo cómo número sin signo, este será siempre superior a 8 (valor de DIM). Las instrucciones resultantes fueron:

```
cmp r1, #DIM

cmplo r2, #DIM

bhs ficha_valida_arm_else
```

Luego para realizar el segundo bloque ((tablero[f][c] != CASILLA_VACIA)) se usaron 4 instrucciones. Primero utilizando rotaciones se consiguió sin multiplicar obtener la posición relativa al inicio del tablero en la que se encuentra tablero[f][c]. Luego se comparó el contenido de esta con CASILLA_VACIA. Las instrucciones resultantes fueron:

```
add r4, r2, r1, LSL #LOG2_DIM

ldrb r0, [r0, r4]

cmp r0 , #CASILLA_VACIA

beq ficha_valida_arm_else
```

La traducción de las sentencias que se ejecutan dentro del bloque if y el else son traducciones literales desde c.

Durante toda la función se logró trabajar con los registros del 0 al 3, de esta forma se evitó guardar registros.

**Código:**

Para evitar recargar este documento, la implementación de la función comentada se encuentra en el fichero init_b_2017.asm

## 5.2 Función ficha válida THUMB

Para el diseño de la función ficha válida THUMB, se usó una pasarela, la cual cambia el modo de procesador a THUMB y llama a la función propiamente dicha.

El diseño de esta función sigue la misma estructura que el de ficha válida, aunque al no poderse utilizar ejecución condicional, el primer bloque de la sentencia if se realizó en 4 instrucciones.

El segundo bloque al no poderse realizar rotaciones en la instrucción ADD y tener que hacerla aparte, también utiliza una instrucción más.

**Código:**

Para evitar recargar este documento, la implementación de la función comentada se encuentra en el fichero init_b_2017.asm

## 5.3 Función patrón volteo ARM

En el caso de la función patrón volteo, primero se cargan las variables pasadas por pila, utilizando la instrucción LDRB, no se utiliza load múltiple para cargarlas, y esto es debido a que con load múltiples no se puede cargar bytes y los parámetros que se pasan por pila son caracteres.

Como esta función llama a otras funciones, en el bloque de activación se guarda el link register y también los argumentos que recibe se guardan en registros superiores al 3. Con esto se evitar tener que guardarlos al llamar a otra función.

Al hacer la llamada a ficha válida, a los argumentos SA y CA se les aplica la máscara 0xff, debido a que anteriormente se operó con ellos y podrían haber generado un valor superior al almacenable en un carácter. Las instrucciones resultantes fueron:

and r1, FA, #0xff

and r2, CA, #0xff

Como a ficha válida se le pasa un entero por referencia, el cual es declarado en patrón volteo, se reserva una palabra en la pila y se le pasa la dirección de esta.

Para permitir llamar a las diferentes funciones ficha válida (C, ARM y THUMB), en vez de llamarse a una en concreto, se llama a un puntero a función el cual apuntará a la función ficha válida que se desea ejecutar. Este puntero está declarado en el fichero test.c y es el que permite realizar las pruebas sobre todas las combinaciones de forma automática.

Para comprobar si después de la llamada a ficha_valida se entra en el bloque if, else if o else, sólo se usan dos comparaciones ya que si posición válida es distinto de 1 entrará en el bloque else, en cambio sí es 1, la entrada al bloque if o el else if depende de si casilla es igual o distinto de color. Las instrucciones resultantes fueron:

cmp  posicion_valida, #1

bne patron_volteo_arm_else

cmp casilla, color

beq patron_volteo_arm_else_if

# Inicio bloque if

En el bloque if se realiza una llamada recursiva, por lo que se vuelve a realizar una máscara con 0xff al pasar los parámetros que son caracteres y puedan ocupar más de un byte por haber operado con ellos.

En el bloque else if se sustituye el bloque if anidado por una ejecución condicional en la cual se asigna a r0(registro que almacena el resultado de la función) el valor correcto. Las instrucciones resultantes fueron:

cmp longitud, #0

movgt r0, #PATRON_ENCONTRADO

movle r0, #NO_HAY_PATRON

El bloque else es una traducción literal del código en c.

Por último, al restablecer los parámetros desde la pila, se carga directamente el link register anterior en pc, con esto se evita tener que hacer un salto a la posición apuntada por el link register.

**Código:**

Para evitar recargar este documento, la implementación de la función comentada se encuentra en el fichero init_b_2017.asm

# 6. Resultados

## 6.1 Diferencia de rendimiento y tamaño del código para las posibles combinaciones de las funciones:

Para obtener la diferencia de rendimiento y tamaño del código para las posibles combinaciones de las funciones ficha válida y patrón volteo (C-ARM, C-THUMB, C-C, ARM-ARM, ARM-THUMB, ARM-C) se utilizó un bucle que realiza un millón de iteraciones con cada una de las distintas combinaciones y se cronometró el tiempo de ejecución de cada una en los ordenadores del laboratorio.

**Resultados:**

C-ARM

| Nivel optimización | Número instrucciones ficha_valida | Número instrucciones patron_volteo | Tamaño código (bytes) | Número instrucciones ejecutadas | Tiempo ejecución (segundos) |
|---|---|---|---|---|---|
| O0 | 15 | 69 | 336 | 45 | 5,5 |

C-THUMB

| Nivel optimización | Número instrucciones ficha_valida | Número instrucciones patron_volteo | Tamaño código (bytes) | Número instrucciones ejecutadas | Tiempo ejecución (segundos) |
|---|---|---|---|---|---|
| O0 | 23 | 69 | 322 | 53 | 6,1 |

C-C

| Nivel optimización | Número instrucciones ficha_valida | Número instrucciones patron_volteo | Tamaño código (bytes) | Número instrucciones ejecutadas | Tiempo ejecución (segundos) |
|---|---|---|---|---|---|

| | | | | | |
|---|---|---|---|---|---|
| O0 | 45 | 69 | 456 | 68 | 7,1 |

ARM-ARM

| Nivel optimización | Número instrucciones ficha_valida | Número instrucciones patron_volteo | Tamaño código (bytes) | Número instrucciones ejecutadas | Tiempo ejecución (segundos) |
|---|---|---|---|---|---|
| O0 | 15 | 43 | 232 | 33 | 4,7 |

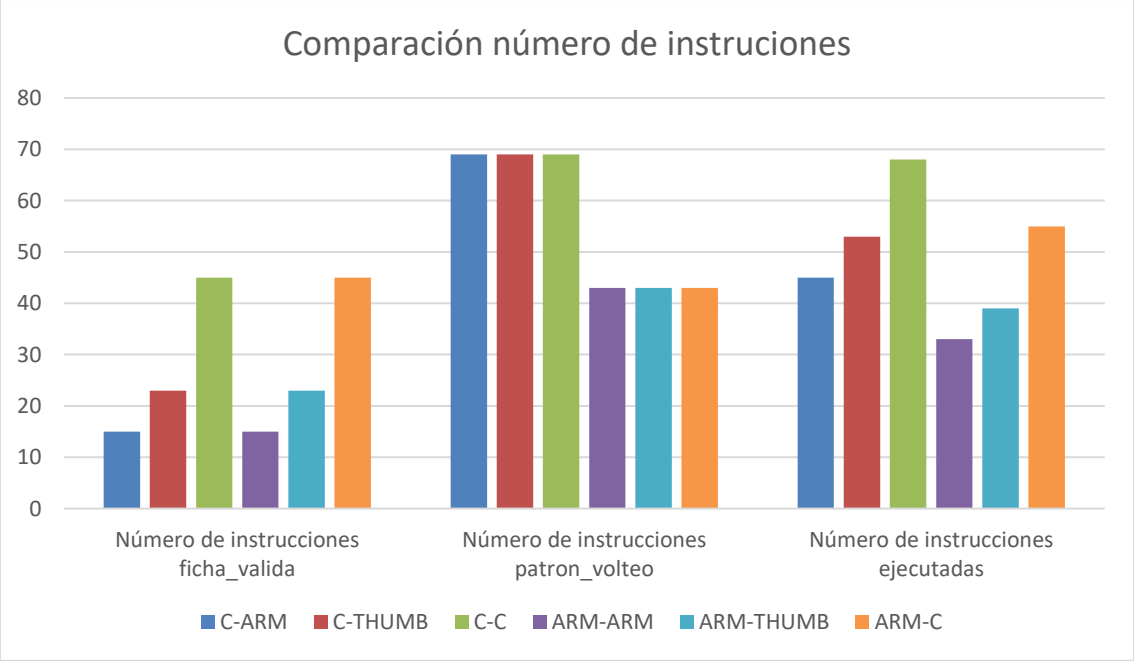
ARM-THUMB

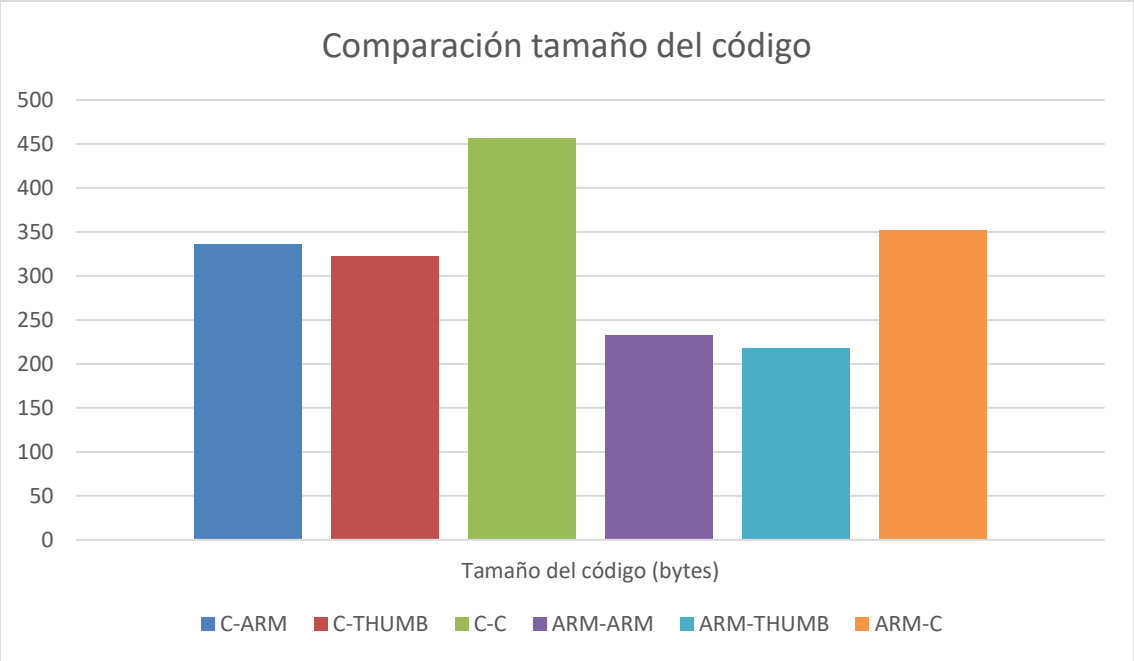
| Nivel optimización | Número instrucciones ficha_valida | Número instrucciones patron_volteo | Tamaño código (bytes) | Número instrucciones ejecutadas | Tiempo ejecución (segundos) |
|---|---|---|---|---|---|
| O0 | 23 | 43 | 218 | 39 | 5,3 |

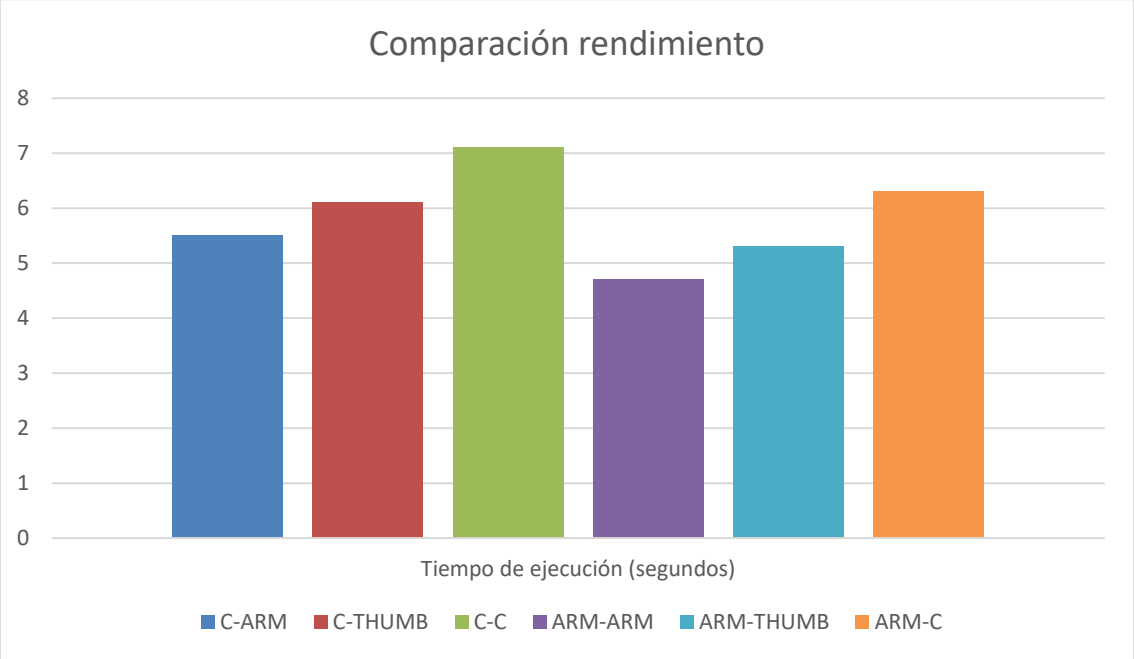
ARM-C

| Nivel optimización | Número instrucciones ficha_valida | Número instrucciones patron_volteo | Tamaño código (bytes) | Número instrucciones ejecutadas | Tiempo ejecución (segundos) |
|---|---|---|---|---|---|
| O0 | 45 | 43 | 352 | 55 | 6,3 |

**Graficas comparativas:**






Como se puede observar, la combinación que mejor rendimiento presenta es la combinación ARM-ARM, aunque como es previsible la que menos espacio ocupa en memoria es la combinación ARM-THUMB.

## 6.2 Diferencia de rendimiento y tamaño del código para los distintos grados de optimización del compilador en la combinación C-C:

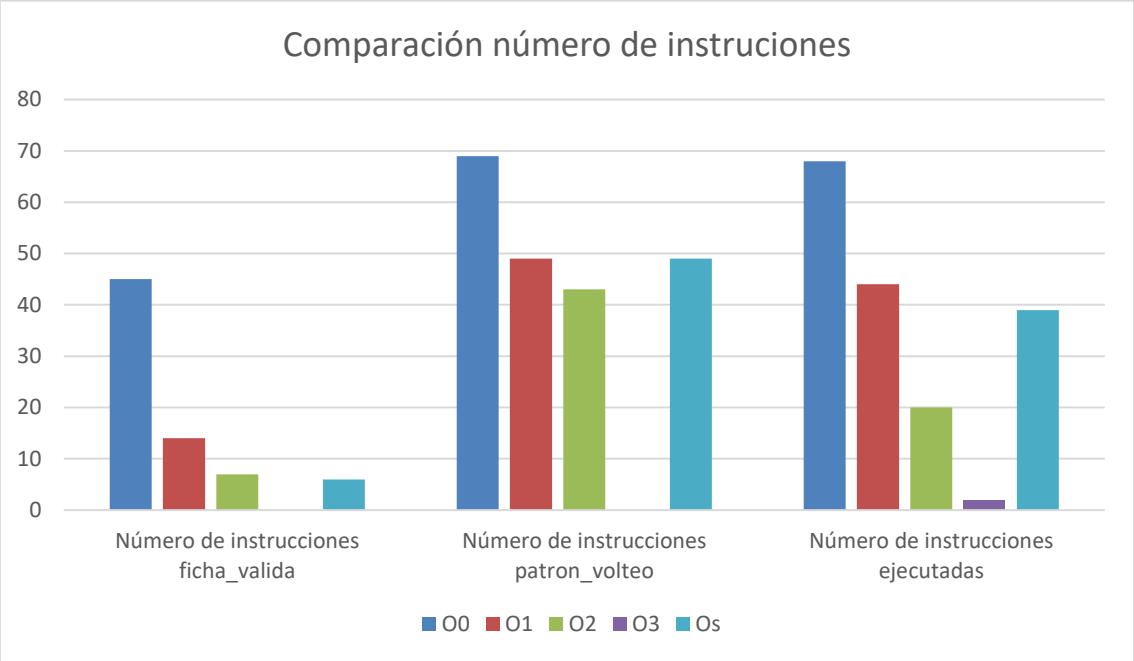
Para medir el número de instrucciones en cada grado de optimización de las funciones en C ficha válida y patrón volteo se ha procedido mediante el modo depuración, observando el desensamblado. Para obtener el rendimiento se ha ejecutado en un bucle de un millón de iteraciones. Al haberse medido en una máquina diferente a la que se usó para medir las distintas combinaciones de las funciones, los resultados no serían directamente comparables, aunque viendo el tiempo de ejecución en la otra máquina de la combinación C-C y el tiempo de ejecución en esta de esa misma combinación con optimización o0 nos podemos hacer una ligera idea del cambio entre máquinas.

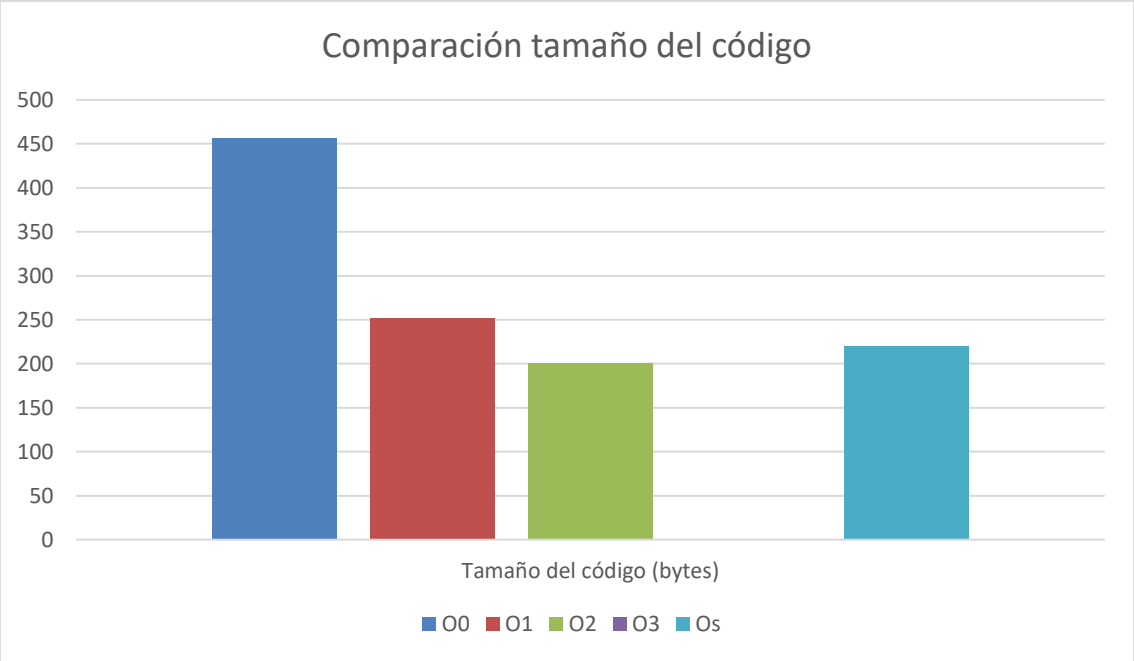
**Resultados:**

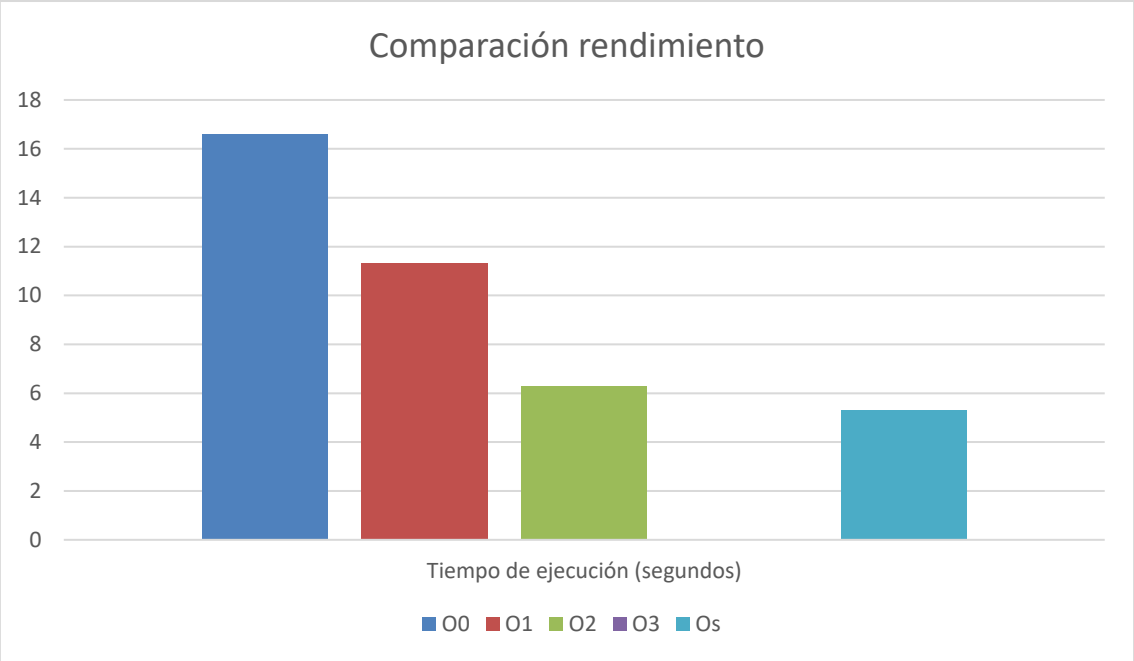
Pruebas C-C

| Nivel optimización | Número instrucciones ficha_valida | Número instrucciones patron_volteo | Tamaño código (bytes) | Número instrucciones ejecutadas | Tiempo ejecución (segundos) |
|---|---|---|---|---|---|
| O0 | 45 | 69 | 456 | 68 | 16,6 |
| O1 | 14 | 49 | 252 | 44 | 11,3 |
| O2 | 7 | 43 | 200 | 20 | 6,3 |
| O3 | 0 | 0 | 0 | 2 | 0 |
| Os | 6 | 49 | 220 | 40 | 11,4 |

**Graficas comparativas:**






Se puede comprobar cómo con las diferentes optimizaciones la función cambia su tiempo de ejecución y número de instrucciones ejecutadas, en las métricas relativas al tiempo, se puede comprobar cómo se reduce el tiempo de ejecución conforme vamos aumentando los grados de optimización.

Con el nivel de optimización O0 produce un código bastante ineficiente el cual sólo mantiene en registro los valores con los que está operando, esto hace la depuración más sencilla a costa de tener que realizar muchas llamadas a memoria.

En el nivel de optimización O1 el compilador utiliza todos los registros, de esta forma evita accesos innecesarios a memoria, además empieza a realizar optimizaciones sobre el código. Por ejemplo, en la función ficha válida aprovecha el hecho de que, en un número negativo, al menos el último bit ha de estar a 1, por lo que, si lo interpretas como un número sin signo, su valor será muy elevado. Si este número es de tipo carácter y quieres comprobar si se encuentra en el rango [0, 7] sólo has de hacer una comparación si es mayor a 7 interpretando el número como si no tuviese signo. Si sale falso está en el intervalo (esta optimización la hemos aplicado a nuestro código).

En el nivel de optimización O2 aparte de las mejoras anteriores se realiza un inline de la función ficha válida, lo cual evita hacer la llamada a la función, reduciendo de esta forma saltos y evitando el paso de parámetros.

En el nivel de optimización O3, directamente precalcula el resultado, asignando a las variables que modifica la función, su resultado después de aplicarla, por ello hemos asumido un tamaño del código de 0 instrucciones.

En el nivel de optimización OS se presentan las mismas características que en el o1, aunque el código resultante es diferente, ligeramente más contenido gracias a la reducción de la función ficha válida.

## 6.3 Diferencia de rendimiento y tamaño del código para los distintos grados de optimización del compilador en la combinación C-ARM:

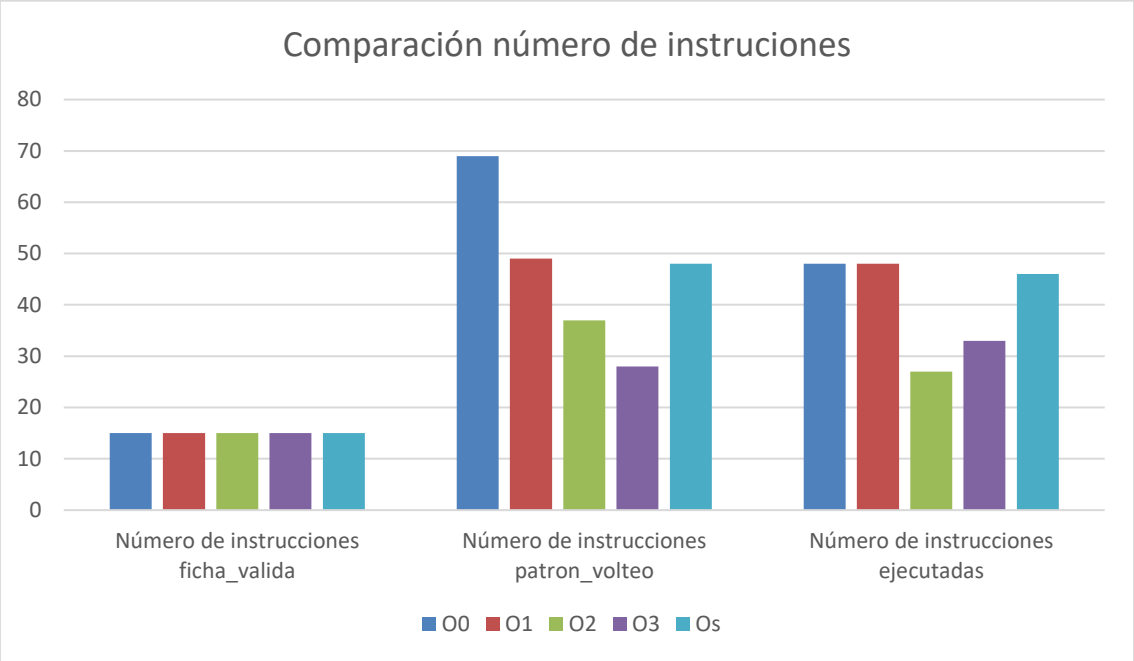
Para medir el número de instrucciones en cada grado de optimización de las funciones en ficha válida ARM y patrón volteo se ha procedido mediante el modo depuración, observando el desensamblado. Para obtener el rendimiento se ha ejecutado en un bucle de un millón de iteraciones.

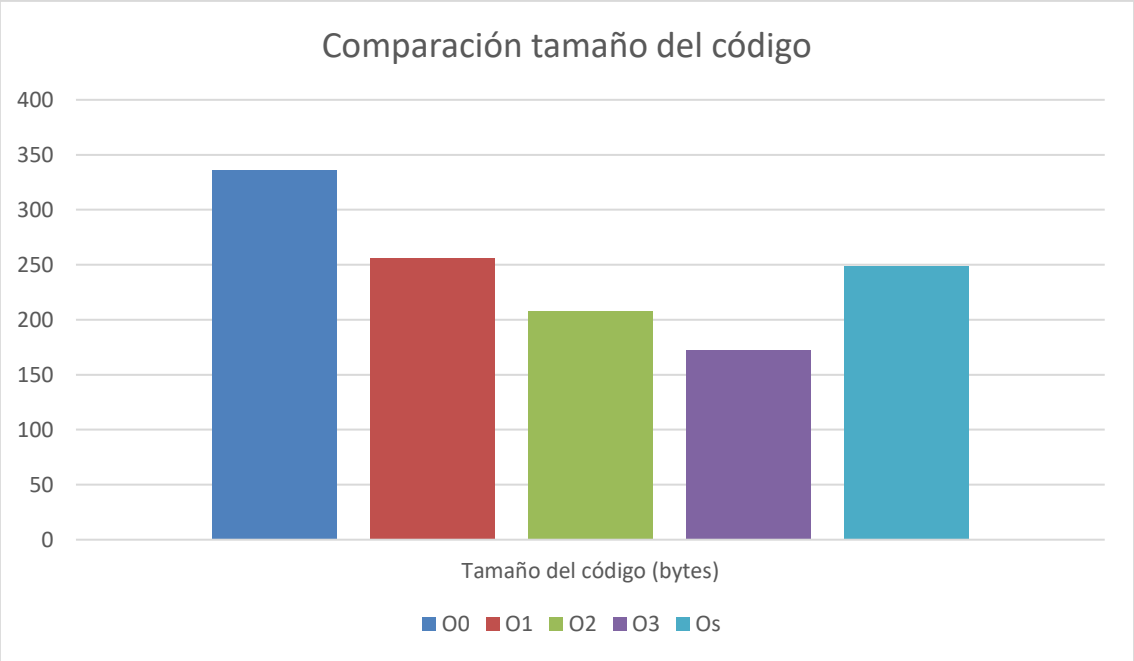
**Resultados:**

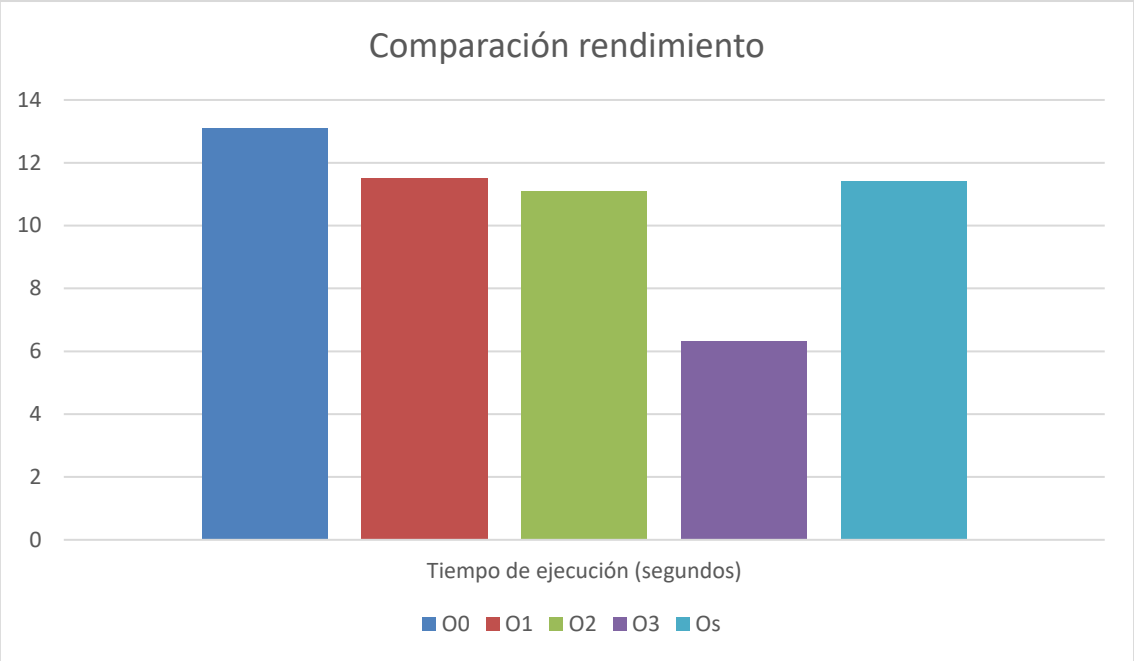
Pruebas C-ARM (función bucle)

| Nivel optimización | Número instrucciones ficha_valida | Número instrucciones patron_volteo | Tamaño código (bytes) | Número instrucciones ejecutadas | Tiempo ejecución (segundos) |
|---|---|---|---|---|---|
| O0 | 15 | 69 | 336 | 48 | 13,1 |
| O1 | 15 | 49 | 256 | 48 | 11,5 |
| O2 | 15 | 37 | 208 | 27 | 11,1 |
| O3 | 15 | 28 | 172 | 33 | 6,3 |
| Os | 15 | 48 | 252 | 46 | 11,7 |

**Graficas comparativas:**






Se puede comprobar cómo el número de instrucciones de patrón volteo reduce conforme aumentamos el grado de optimización. Pero en cambio, el número de instrucciones ejecutadas varía muy poco. Esto es debido a que ficha válida siempre cuenta con el mismo número de instrucciones, y patrón volteo, a pesar del estado de optimización, ha de ejecutar ese número de instrucciones para completar la función.

En términos de rendimiento, tanto en el caso de instrucciones ejecutadas como tiempo de ejecución, es en el nivel de optimización O3 en la que más diferencia se aprecia.

# 7. Pruebas automáticas

Para comprobar de una forma automática el correcto funcionamiento de nuestras funciones aisladas en ARM y THUMB, además de probar también las distintas combinaciones entre C-C, C-ARM, C-THUMB, ARM-C, ARM-ARM y ARM-AHUMB, se han realizado dos funciones de prueba: test_ficha_valida y test_patron_volteo.

En test_ficha_valida se declaran varios casos de prueba de distintas posiciones del tablero y con estos datos se ejecutan las funciones ficha_valida, ficha_valida_arm y ficha_valida_thumb, guardando sus resultados y comparando que sean iguales. Si falla THUMB o ARM se activará un flag (FALLA_ARM y FALLA_THUMB) que indica que ha fallado, así como la variable PRIMER_FALLO contendrá el número del caso de prueba en el que ha fallado por primera vez. Una vez comete el primer fallo, ya no sigue probando otros casos.

En test_patrón_volteo, además de comprobar que la función está bien en ARM, se prueban también las distintas combinaciones entre funciones nombradas anteriormente. Al igual que en test_ ficha_valida, se declaran al principio varios casos de prueba, en este caso son tres tableros del reversi con unas posiciones de fichas determinadas puestas especialmente para

que haya varios patrones en cada caso. Se llama a todas las combinaciones con el mismo caso de prueba y se almacena su resultado en unas variables distintivas, comprobando al final que todos los resultados coincidas. También trabaja con flags que ayudan a saber si falla, qué falla y dónde.

Lo que se trata con estas pruebas es automatizar la corrección de las funciones para que, si posteriormente se modifican, se tenga un método de verificar su funcionamiento sin tener que estar depurando y declarando casos constantemente, haciendo bastante más cómoda la labor de la corrección.

**Código:**

El código de estas pruebas se puede encontrar en tests.c.

# 8. Conclusiones

Tras la realización del proyecto queda claro como el código ensamblador es una excelente técnica de optimización del código, aunque no logra igualar los resultados del compilador con nivel de optimización o3, entre otras cosas, debido a que este último no respeta el modularidad del código, y precalcula los resultados.

De los tiempos también se puede extraer que el modo de ejecución THUMB es algo menos eficiente que el ARM, aun así, es una buena opción teniendo en cuenta el ahorro en almacenamiento que supone.

Con respecto a los grados de optimización, cabe destacar lo agresivos que resultan los grados O2 y O3, ya que al eliminar el código no útil (es decir, las asignaciones, operaciones o llamadas a funciones que no tienen una repercusión externa se eliminan del código), hace difícil depurarlo, ya que dependiendo del contexto de la llamada a la función produce códigos muy diferentes.

Queda claro también la mejora que supone al menos utilizar el grado de optimización O1 frente a la utilización del O0, en el cual se producía un número muy elevado de accesos a memoria.