Proyecto Hardware

# Práctica 1: Desarrollo y optimización de código para el procesador ARM

# Proyecto Hardware

# 2017/18

# 

# Jorge Aznar López - 721556

# Alex Oarga Hategan - 718123

Índice

* 1 Introducción Página
* 2 Resumen Página
* 3 Objetivos Página
* 4 Metodología Página
* 5 Resultados Página
* 6 Conclusiones Página
* Bibliografía Página

# 1. Resumen

# 2. Introducción

Se van a desarrollar en código ARM y THUMB, para el procesador ARMv7, las funciones "ficha válida" y "patrón volteo" (sólo en ARM) en base al código proporcionado en lenguaje "C" del Otello (Reversi), para un tablero con dimensiones establecidas de 8x8.

# 3. Objetivos

El primer objetivo del proyecto es la traducción del código proporcionado en C a lenguaje ensamblador ARM y THUMB. Como la función "patrón volteo" invoca a "ficha válida", se pretende desarrollar varias combinaciones de las funciones. Estas combinaciones serán (patron\_volteo - ficha\_valida): C-C, C-ARM, C-THUMB, ARM-C, ARM-ARM, ARM-THUMB. Posteriormente, se desarrollará un repertorio de pruebas para evaluar el correcto comportamiento de cada una de las combinaciones. Finalmente, se evaluarán las prestaciones de dichas combinaciones y se compararán entre ellas, además de con diferentes versiones de optimización del código en C (-O1, -O2,-03 y -0s).

# 4. Metodología

4.1. Estructura del proyecto

4.1.1 Ficheros

Los ficheros que contiene el proyecto son los siguientes:

[CAPTURA]

4.1.2 Funciones

Se van a desarrollar en el fichero 'Apartado2.asm' las funciones 'ficha\_valida' y 'patron\_volteo' que se incluyen en el código C original del juego reversi8. Las versiones que se van a desarrollar en este fichero son:

la versión ARM de la función ficha\_valida(), la versión thumb de ficha\_valida() y la versión ARM de patron\_volteo(). Patron\_volteo() es una función que llama a su vez a la función ficha\_valida, así que las 3 versiones que se van a desarrollar y el código en c dado, se van a utilizar para las diferentes combinaciones de llamadas.

Como el objetivo es generar las 6 combinaciones: C-C, C-ARM, C-THUMB, ARM-ARM, ARM-C, ARM-THUMB; se va a desarrollar una función para cada una de las combinaciones que se organizaran de la siguiente manera:

- en reversi8\_2017.c: las funciones 'patron\_volteo\_c\_c' (las funciones originales sin modificar), 'patron\_volteo\_c\_arm' y 'patron\_volteo\_thumb' (las funciones originales variando la llamada a ficha\_valida() según cada caso).

- en Apartado2.asm: 'patron\_volteo\_arm\_arm' (versión ARM de patron\_volteo que llama a la versión ARM de ficha\_valida), 'patron\_volteo\_arm\_c' (versión ARM de patron\_volteo que llama a la versión C de ficha\_valida) y 'patron\_volteo\_arm\_thumb' (versión ARM de patron\_volteo que llama a la versión thumb de ficha\_valida), además de las propias funciones ficha\_valida\_arm y ficha\_valida\_thumb.

4.1.2 Líneas de código

El total de líneas de código que se ha obtenido en este proyecto (sin contar comentarios, apertura o cierre con '{''}') es el siguiente:

reversi8\_2017.c

ficha\_valida (función original): 9

patron\_volteo\_c\_c (función original): 18

patron\_volteo\_c\_arm: 18 (solo cambia la llamada a ficha\_valida)

patron\_volteo\_c\_thumb: 18 (solo cambia la llamada a ficha\_valida)

Apartado2.asm

ficha\_valida\_arm: 12 instrucciones ARM

ficha\_valida\_thumb: 20 instrucciones thumb

patron\_volteo\_arm\_arm: 39 instrucciones ARM

patron\_volteo\_arm\_c: 39 instrucciones ARM

patron\_volteo\_arm\_thumb: 39 instrucciones ARM

4.2 Desarrollo

4.1 ficha valida arm

La función ficha valida recibe 4 argumentos. Se sigue el estándar ATPCS por lo que al ser 4 parámetros se pasaran en los registros r0-r3. Los 3 primeros caracteres que se reciben son de tipo char. Según el estándar estos datos (que ocupan 1 byte) se rellenan con 0s hasta llenar los 32 bits. El primer parámetro es un vector de char así que no se reciben los datos sino la dirección donde comienza el vector. El cuarto parámetro es un entero (int) que se pasa como un puntero, por lo que se recibirá como parámetro la dirección de donde esta contenido.

Lo primero que se ejecuta en la función es el condicional if. Para compara que f y c (parámetros r1 y r2) son mayores o iguales que 0 y menores que 8 (DIM), se va a utilizar la instrucción CMP e instrucciones predicadas. En este caso se ha comparado con 7. De esta forma se puede utilizar los condicionales sin signo HI y LS para comparar a la vez que el valor sea mayor o igual a 0 y menor que 8, ya que si el valor es negativo, si se considera que es un numero sin signo, será mayor que el propio valor. \*\* REVISAR \*\*

Se realiza la primera comparación (con el valor de f) y en caso de cumplir las condiciones se evalúa la segunda condición (con el valor de c). En caso de ser mayor sin signo, con el condicional HI se salta con la instrucción B a la sección correspondiente al código else, identificado con la etiqueta 'else\_fvarm'. En caso de cumplir la condición continua la ejecución del código sin saltar.

ADD r0, r0, r1, lsl#3

Si continua la ejecución, se pasa a comparar tablero[f][c] con CASILLA\_VACIA (0). El vector tablero contiene los datos (\*\*\*\*\*\*\*\*\*\*\*\*\*\*). Por lo tanto, para acceder al valor [f][c] se tendrá que calcular la posición correspondiente a: @tablero + f\*DIM + c (\*\*\*\*\*\*\*\*\*\*\*\*), siendo DIM en este caso 8. Para multiplicar por 8, se ha utilizado el operador de desplazamiento lsl 3, pues multiplicar por 8 (2^3) un numero binario es equivalente a desplazar el numero 3 bits a la izquierda. Este operador se utiliza en la operación add que suma a la dirección de tablero el valor f al que se le aplica el desplazamiento, es decir:

ldrb r0, [r0, r2]

Para cargar la posición [f][c] todavía es necesario sumarle c. Esto se consigue con el operador de offset de la instrucción LDR, para el cual se utiliza el registro de c (r2). Es decir, la operación carga en r0 la dirección r0+r2. Como el dato que se carga es un tipo char, se utiliza la instrucción ldrb para cargar solo la longitud de un byte de memoria en el registro.

cmp r0, #0

bne .if\_fv\_arm

Por último para concluir la instrucción if, se compara el resultado cargado con CASILLA\_VACIA con la instrucción CMP y en caso de no ser igual (cumple la condición de if), se ejecuta la instrucción de salto B con el condicional NE (si no es igual) a la etiqueta if\_fv\_arm que se corresponde con el código del cuerpo de if. En caso contrario, si no se cumple la condición de if, se entra directamente el cuerpo correspondiente al else por estar el código de este seguido.

mov r0, #1 \*\*\*\*\* REVISAR CODIGO IMPORTANTE\*\*\*\*\*

str r0, [r3, #0]

La función ficha\_valida solo devuelve un tipo de dato entero (int) por lo que lo devolverá en el registro 0. Como también modifica la variable posicion\_valida que se ha pasado como parámetro por referencia (dirección en memoria), habrá que escribir su nuevo valor en memoria.

En el cuerpo de if, se actualiza posición valida con el valor de ??????. Por eso se mueve a r0 dicho valor y se escribe en la dirección de posicion\_valida, hasta ahora contenida en r3. Se tiene que devolver el valor tablero[f][c], que es el que ya contiene en r0.

mov r0, #0 \*\*\*\*\* REVISAR CODIGO IMPORTANTE\*\*\*\*\*

str r0, [r3, #0]

En el caso de else, debe asignar 0 a la variable posicion\_valida y devolver también 0 (CASILLA\_VAICA). Se carga por ello el valor 0 en r0, se escribe en la dirección de posicion\_valida y al estar en r0 es el valor que devolverá la función.

BX r14

Por último, tanto en if como en else, para volver a la dirección desde la que se invoco a la función (contenida en LR), se ejecuta la anterior instrucción. La instrucción BX cambia también el estado entre thumb o ARM, en caso de que la función hubiese sido invocada desde thumb.

4.1 ficha valida thumb

En cuanto a planteamiento, la función ficha\_valida\_thumb es similar a ficha\_valida\_arm salvo por algunas diferencias que se comentan a continuación:

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* push \*\*)

thumb no permite la comparación con inmediatos por lo que hemos tenido que utilizar un registro extra. EL registro r4, al ser para variables privadas, se debe guardar en la pila antes de modificarlo y se debe restaurar al terminar.

MOV r4, #(\*\*\*\*\*\*\*\*\*\*\*DIM-1)

Se carga en r4, DIM-1 para poder usarlo en la instrucción CMP-

BHI .Elsethumb

En caso de ser mayor sin signo (como en ficha\_valida\_arm), se salta a la etiqueta correspondiente con el código de else.

lsl r4, r1, #3

Como no se puede aplicar el desplazamiento junto a otra instrucción, como en la función anterior, se utiliza la función de thumb lsl que realiza dicha operación.

add r0, r0, r4

ldrb r4, [r0, r2]

cmp r4, #0

beq .Elsethumb

Se carga el valor y se compara igual que en la función anterior. En caso de no cumplirse la condición tablero[f][c] salta al código de else con beq. En caso contrario ejecuta el código de if por estar seguido.

Las operaciones que se realizan dentro de if y else son las mismas que en la función anterior.

(\*\*\*\*\*\*\*\*\*\*\*pop)

BX r14

Al terminar la función, se desapila el registro r4 y se vuelve a la dirección invocante(r14/lr) cambiando el estado de thumb a ARM si se invocase desde ARM.

4.1 patrón volteo arm

La función recibe un total de 7 parámetros. Como solo se pueden utilizar los registro de r0 a r3 para parámetros, recibe los 3 restantes en la pila. Los parámetros que recibirá son: r0:@tablero, r1:@longitud, r3:char FA, r4:char FC, pila: char SF, char SC, char color.

( \*\*\*\*\*\*\*\*\*\*\* MARCO PILA \*\*\*\*\*\*\*\*\*\*\*\*)

SUB sp, sp, #4

Se guarda espacio en la pila para la variable local posicion\_valida.

MOV r7, r1

MOV r8, r0

Patron\_volteo va a tener que invocar a ficha\_valida. Los parámetros que recibe ficha valida son: r0:@tablero, r1:f, r2:c, r3:@posicion\_valida. Es por eso que tablero y longitud se guardan en r8 y r7 respectivamente para utilizarse luego.

( \*\*\*\*\*\*\*\*\*\*\* AND)

SUB r3, fp, #32

Se almacena en r3 la dirección de la variable local posicion\_valida para la que se reservo espacio al principio. Esta dirección se pasara como parámetro a ficha\_valida (posicion\_valida se pasa por referencia). Para acceder a la variable local se resta a fp el número de elementos apilados al principio\*4.

MOV r6, r1

MOV r9, r2

Las variables FA y FC se podrían utilizar luego en la función así que salvan en los registro r6 y r9.

BL ficha\_valida\_arm

Se llama a la función ficha\_valida con la instrucción BL. En el caso de llamar a ficha\_valida\_thumb, como tendría que cambiar de estado, se debe llamar con la instrucción BX.

MOV r3, r9

MOV r2, r6

MOV r1, r7

LDR r6, [fp, #-32]

Se restauran los parámetros salvados (tablero no porque todavía no es necesario) y se carga en r6 el valor que ha escrito ficha\_valida en la dirección de posicion\_valida.

CMP r6, #1

BNE else\_pv

Para evaluar el condicional if, primero se comprueba la condición de si posicion\_valida es igual a (\*\*\*\*\*\*\*\*). En caso de que no lo sea se salta con la instrucción BNE a la parte del código correspondiente a else.

LDRB r6, [fp, #12] // cargamos color de la pila

LDR r7, [r1] // se carga \*longitud (se utiliza después)

CMP r6, r0 // IF color==casilla

BEQ elseif\_pv

Una vez comprobado lo anterior, si no se ha realizado el salto, se carga el ultimo parámetro de la pila y se compara con casilla (resultado de ficha\_valida en r0). En caso de ser iguales, entra dentro de la condición else if (la estructura condicional del código en c consta de if, else if y else). En caso contrario, el código entra al código de if por estar seguido.

ADD r7, r7, #1

STR r7, [r1]

MOV r0, r8 // tablero a r0 para llamar a patron\_volteo

STMFD sp!, {r4-r6} // apilamos sf, sc, color

BL patron\_volteo // patron\_voleto()

ADD sp, sp, #12 // desapilamos los parámetros

Dentro del código de if, se aumenta el valor de longitud en 1. Como guardamos longitud mediante su dirección es necesario escribir en memoria el nuevo valor. Después de esto, se mueve tablero a r0 y se apilan SF, SC y color porque se va a llamar de forma recursiva a patrón volteo. Una vez termina la función, se desapilan los 3 parámetros apilados (3\*4).

La función patron\_volteo devuelve un entero (en r0). En el caso de if, devolverá el mismo resultado que devuelve la llamada recursiva a patron\_volteo, valor que ya se encuentra en r0.

CMP r7, #0 // devuelve el resultado según el valor de longitud

MOVGT r0, #1

MOVLE r0, #0

En el caso de else if, se compara longitud con 0, y en función del resultado, patrón volteo devuelve en r0 el valor 0 o 1.

MOV r0 #0

En el caso de else, se devuelve 0 en r0.

(\*\*\*\*\*\*\*\*\* DESAPILAR)

4.3 Optimización

4.3.1 ficha valida arm

ficha\_valida\_arm

cmp r2, \*\*\*\*\*\*\*\*\*\*\*\*\*

cmpls r1, \*\*\*\*\*\*\*\*\*\*\*\*\*

bhi .else\_fv\_arm

Para evitar la comparación con 8 y luego con 0, se aprovechan los predicados condicionales sin signo para comprobar a la vez que sea mayor que 0 y menor que 8. Los predicados se utilizan en la segunda y tercera instrucción que solo se evalúan si se cumple la condición.

add r0, r0, r1, lsl #3 // multiplica por 8 (DIM)

ldrb r0, [r0, r2]

En lugar de utilizar una instrucción para la multiplicación, como la dimensión del vector es potencia de 2 (8), se utiliza el operador de desplazamiento para la multiplicación. Además mediante el segundo operador (offset) de la instrucción ldr nos ahorramos una instrucción add.

Situando el cuerpo dentro de else seguido de las instrucciones de condición, nos ahorramos una instrucción de salto, pues si se cumplen las condiciones no saltara y entrara en esta parte del código.

Para minimizar el uso de registros, se utilizan los registros r0-r3 para las operaciones, evitando así también tener que guardar los registros de variables privadas en memoria.

(\*\*\*\*\*\*\*\*\*\*\* revisar else)

4.3.2 ficha valida thumb

mov r4, #7

cmp r4, r2 // IF c > DIM

bcc .Elsethumb

cmp r4, r1 // IF f > DIM

bcc .Elsethumb (\*\*\*\*\*\*\*\*\*\* hi)

El planteamiento de la comparación es igual que en la función anterior, pero como la instrucción CMP en thumb no tiene operador condicional, es necesario ejecutar los 2 condicionales y en lugar de 2 instrucciones CMP y una B como en el caso anterior, tenemos 2 CMP y 2 B.

lsl r4, r1, #3 // f \* 8(DIM)

Como en la función anterior, se ha utilizado el operador de desplazamiento para la multiplicación con la diferencia de que en thumb no se ha podido integrar en la instrucción ADD.

Tanto en ficha\_valida\_arm como en la función thumb, cargar de memoria el valor de tablero[f][c] se ha dejado para el final de la condición, de forma que si no cumplen las 4 condiciones anteriores no es necesario cargar de memoria el dato.

4.3.3 ficha valida arm

CMP r7, #0 // devuelve el resultado según el valor de longitud

MOVGT r0, #(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

MOVLE r0, #0

Se puede destacar el uso de condicionales para evaluar el valor de longitud.

CMP r6, #1

BNE else\_pv

LDRB r6, [fp, #12] // cargamos color de la pila

LDR r7, [r1] // se carga \*longitud (se utiliza después)

CMP r6, r0 // IF color==casilla

BEQ elseif\_pv

La evaluación de la condición if se divide en dos partes. Primero se comprueba el valor de posicion\_valida y posteriormente se compara casilla con color para determinar si entra en el bloque if o en el bloque else if. Como en los 2 bloques de código anteriores se utiliza el valor de longitud, se carga antes de evaluar la condición para evitar cargar el valor posteriormente en 2 partes del código.

4.4 Problemas encontrados

A continuación se enumeran algunos de los principales problemas que han surgido durante el desarrollo del proyecto:

- Mala creación del bloque de pila que resultaba en accesos incorrectos a los parámetros.

- Errores que surgen solo en la depuración del código optimizado. Esto obliga a la depuración paso por paso de todo el código.

- Apilar y desapilar un numero diferente de recursos que resulta en errores de ejecución.

- Errores entre las llamadas de arm al código c.

- El planteamiento del parámetro por referencia posicion\_valida.

- Incorrecta traducción del código C a ARM o THUMB

- Suma de tipos char

4.5 Reparto del trabajo

# 5. Resultados

## 5.1. Resultados obtenidos tras el análisis del tamaño del código

Analizando el código generado por el compilador para las funciones "ficha válida" y "patrón volteo" en el lenguaje C a partir del desensamblado, pues las demás son implementadas, hemos obtenido las siguientes métricas para el número de instrucciones y, proporcionalmente a estas, el tamaño en bytes del código.

A cerca del número de instrucciones:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Instrucciones | pv\_c | pv\_arm | fv\_c | fv\_arm | fv\_thumb |
| O0 | 50 | 39 | 14 | 12 | 20 |
| O1 | 50 | 39 | 14 | 12 | 20 |
| O2 | 47 | 39 | 14 | 12 | 20 |
| O3 | 53 | 39 | 14 | 12 | 20 |
| Os | 47 | 39 | 13 | 12 | 20 |

A cerca del número de bytes:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Bytes | pv\_c | pv\_arm | fv\_c | fv\_arm | fv\_thumb |
| O0 | 200 | 156 | 56 | 48 | 40 |
| O1 | 200 | 156 | 56 | 48 | 40 |
| O2 | 188 | 156 | 56 | 48 | 40 |
| O3 | 212 | 156 | 56 | 48 | 40 |
| Os | 188 | 156 | 52 | 48 | 40 |

Se ha realizado una comparación del rendimiento entre los diferentes niveles de optimización para el código generado, sin grandes singularidades entre ellos:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Rendimiento | pv\_c | pv\_arm | fv\_c | fv\_arm | fv\_thumb |
| O0 - O1 | 100% | 100% | 100% | 100% | 100% |
| O0 - O2 | 106% | 100% | 100% | 100% | 100% |
| O0 - O3 | 94% | 100% | 100% | 100% | 100% |
| O0 - Os | 106% | 100% | 108% | 100% | 100% |
| O1 - O2 | 106% | 100% | 100% | 100% | 100% |
| O1 - O3 | 94% | 100% | 100% | 100% | 100% |
| O1 - Os | 106% | 100% | 108% | 100% | 100% |
| O2 - O3 | 89% | 100% | 100% | 100% | 100% |
| O2 - Os | 100% | 100% | 108% | 100% | 100% |
| O3 - Os | 113% | 100% | 108% | 100% | 100% |

Se puede observar que el nivel de optimización -O3 es aquel que mayor número de instrucciones en ensamblador genera, esto es debido a que al ser un nivel de optimización agresivo en tiempo de ejecución, activa una serie de optimizaciones más costosas en tiempo de compilación y memoria, como la vectorización de bucles dentro del código, aunque en este proyecto apenas se pueda percibir debido al reducido tamaño del código.

Los tamaños en bytes del código se han obtenido multiplicando por cuatro cada instrucción en ARM, y duplicando cada instrucción en THUMB, esto es debido a que las instrucciones de ARM ocupan una palabra entera y las de THUMB media palabra en memoria.

## 5.1. Resultados obtenidos tras el análisis del tiempo de ejecución

Antes de comenzar el análisis de los resultados, cabe destacar que estos se han obtenido ejecutando un millón de veces cada función, lo cual nos da el resultado en segundos, en vez de en microsegundos.

Estas son las métricas obtenidas para cada combinación (C-C, C-ARM, C-THUMB, ARM-C, ARM-ARM, ARM-THUMB) y nivel de optimización:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Tiempos | C-C | C-ARM | C-THUMB | ARM-C | ARM-ARM | ARM-THUMB |
| O0 | 7,42 | 5,38 | 5,80 | 6,60 | 4,60 | 5,20 |
| O1 | 4,71 | 4,67 | 5,10 | 3,94 | 3,77 | 4,37 |
| O2 | 2,67 | 4,24 | 4,75 | 3,50 | 3,51 | 4,05 |
| O3 | 0,57 | 2,13 | 2,66 | 3,57 | 3,60 | 4,20 |

Cabe destacar que en las funciones que no involucran código C, la disminución de los tiempos de ejecución no es tan drástica pero si se nota debido a que también optimiza el bucle "for" generado para el millón de ejecuciones.

Asimismo como en el apartado anterior, se ha realizado una comparación del rendimiento entre los diferentes niveles de optimización para el tiempo de ejecución, en la que si podemos observar claramente que el nivel de optimización "-O3" es el más agresivo en tiempo:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Rendimiento | C-C | C-ARM | C-THUMB | ARM-C | ARM-ARM | ARM-THUMB |
| O0 - O1 | 158% | 115% | 114% | 168% | 122% | 119% |
| O0 - O2 | 278% | 127% | 122% | 189% | 131% | 128% |
| O0 - O3 | 1297% | 252% | 218% | 185% | 128% | 124% |
| O1 - O2 | 177% | 110% | 107% | 113% | 107% | 108% |
| O1 - O3 | 823% | 219% | 192% | 110% | 105% | 104% |
| O2 - O3 | 466% | 199% | 179% | 98% | 98% | 96% |

También se ha decidido incluir una gráfica que refleja claramente el tiempo de ejecución de una manera más visual:

Se puede concluir que -O1 es el nivel de optimización más básico, pues el compilador intentará producir un código rápido y ligero sin tomar mucho tiempo de compilación.

-O2 va paso delante de -O1, pues activará algunas opciones añadidas a las que se activan con -O1. Con -O2, el compilador intentará aumentar el rendimiento del código sin comprometer el tamaño ni el tiempo de compilación.

-O3 es el nivel más alto de optimización posible, como ya se ha comentado previamente, activa optimizaciones que son caras en términos de tiempo de compilación y uso de memoria, el hecho de compilar con -O3 no garantiza una forma de mejorar el rendimiento y, de hecho, en muchos casos puede ralentizar un sistema debido al uso de binarios de gran tamaño y mucho uso de la memoria.

-Os: Optimizará el tamaño del código, útil para máquinas con capacidad de limitada de disco o con CPUs que tienen poca caché.

# 6. Conclusiones

Se han extraído varias conclusiones importantes tras el análisis del código, en primer lugar cabe destacar que un código más corto no siempre implica un menor tiempo de ejecución, esto es debido a que puede que un mayor número de instrucciones que reduzca el número de accesos a memoria, o el uso de instrucciones "costosas" (MUL, p.e.) puede favorecer luego hasta en un 1300% el tiempo de ejecución del código, como se ha podido observar al comparar la versión inicial generada por el compilador sin optimizar frente a la optimizada con el nivel "-O3".

En último lugar, el uso de THUMB en este caso ha dado un resultado favorable, ya que se ha podido comprobar que el coste en tiempo no penaliza lo suficiente como para usar siempre ARM, pues el coste en tiempo solo es un 13% mayor entre ambas, pero THUMB sin embargo ahorra un 20% de memoria.

# Bibliografía