Proyecto Hardware

Práctica 1

27/10/2018

Adrián Samatán Alastuey 738455
Rubén Rodríguez Esteban 737215

Índice de contenidos

1.Resumen

En esta primera práctica de la asignatura se ha procedido a diseñar un juego popular conocido como reversi. Para poderlo programar se han empleado el lenguaje C y el lenguaje ensamblador ARM. Dichos lenguajes han sido trabajados en el entorno de trabajo Eclipse donde se han aplicado las configuraciones necesarias para permitir la compilación cruzada, para poder trabajar así en ambos lenguajes.

Inicialmente se ha partido de un conjunto de archivos de código fuente, escritos en C, y de un conjunto de librerías para poder trabajar con el timer0, un periférico que realiza cada cierto intervalo de tiempo interrupciones sobre la placa con la que se ha trabajado.

En primer lugar se ha entendido el código proporcionado por el profesorado, se ha compilado y generado por medio del compilador de C el código equivalente en lenguaje en ensamblador de las funciones con mayor coste de ejecución, concretamente ficha_valida() y patron_volteo(). Debido a que el código generado por el compilador en ARM es ineficiente se ha procedido a su optimización con la finalidad de reducir el coste de ejecución de ambas.

Para realizar las optimizaciones ha sido necesario recordar la arquitectura del lenguaje ARM, el paso de los parámetros y el gran abanico de instrucciones que ofrece. Así, una vez comprendido todo lo anterior, se ha procedido a elaborar los marcos de pila para indicar cómo se gestionan los parámetros y la memoria reservada en el momento de las llamadas, y además, se han realizado cambios para convertir las instrucciones generadas por el compilador en instrucciones predicadas, para reducir así el tiempo de compilación.

Una vez finalizada esta parte, se ha procedido a realizar las funciones patron_volteo_arm_c() donde se han realizado nuevas optimizaciones con la gestión de los saltos condicionales y el almacenamiento de los parámetros en la pila, ambos cambios están explicados con más detalle en apartados siguientes. Seguidamente se ha elaborado la función patron_volteo_arm_arm() donde se ha realizado el inlining para llamar a ficha_valida() con el fin de eliminar todos los accesos a memoria.

Seguidamente se ha procedido a configurar el timer2 en base a la configuración proporcionada por el profesorado del timer0. Para ello ha sido necesario comprender el funcionamiento del timer0, especialmente el prescalado, cómo poder obtener la máxima precisión a la hora de medir los intervalos y cómo controlar el contenido de los registros para poder gestionar las interrupciones. Una vez realizados estos pasos se ha probado que el código funciona correctamente en la placa y que las interrupciones del timer2 se tratan de manera adecuada.

Posteriormente se han analizado los impactos en el rendimiento que tienen los diferentes niveles de optimización del compilador gcc. Para llevar esta tarea a cabo se han realizado mediciones de todas las funciones características con las que se ha trabajado. Finalmente se han analizado los resultados obtenidos y se han extraído conclusiones acerca de dichos datos, mostradas en apartados posteriores.

2.Introducción

Esta práctica ha consistido en la elaboración de un juego denominado Reversi cuyas reglas de vienen reflejadas en la documentación proporcionada por el profesorado de la asignatura. Para elaborar este trabajo se han empleado dos lenguajes de programación sumamente distintos, concretamente se han utilizado el lenguaje de programación en alto nivel C, y el lenguaje de programación en ensamblador ARM. Adicionalmente se ha trabajado sobre un procesador ARM real, concretamente una placa S3CEV40 sobre la que se han realizado el conjunto de pruebas necesarias para asegurar que el código desarrollado durante la práctica funciona correctamente.

Para poder trabajar con los ficheros de código fuente C se ha usado Eclipse como Entorno de Desarrollo Integrado (IDE). Para poder trabajar con lenguaje C en el entorno Eclipse, se ha tenido que instalar un plugin adicional llamado *CDT*, dado que Eclipse no ofrece por defecto soporte para el lenguaje de programación C. Paralelamente, también se ha trabajado tanto con un manual de ARM, en el que se han podido observar las reglas que componen el diseño de su arquitectura, el formato de las instrucciones y sus tipos, como con manuales acerca de la gestión del sistema de memoria y de entrada y salida de la placa.

Dado que el entorno Eclipse solamente proporciona la interfaz gráfica y la gestión de los proyectos, para poder realizar el resto de tareas especificadas en la práctica se han tenido que emplear herramientas adicionales de GNU, concretamente el compilador *gcc*, el enlazador *ld*, y el depurador *gdb*. Además, se debe tener en cuenta que se ha trabajado sobre un PC para un entorno con procesador ARM, y por tanto se ha necesitado obligatoriamente hacer compilación cruzada. Para distinguir las herramientas cruzadas de las nativas del PC se ha añadido un prefijo que describe la arquitectura objetivo para la que se compila. Dicho prefijo ha sido *arm-none-eabi.*

Como punto de partida, se han empleado los ficheros de código fuente proporcionados por el profesorado de la asignatura y se ha generado un proyecto en Eclipse con ellos. A continuación, se han realizado las configuraciones necesarias para poder compilar correctamente dicho proyecto.

Una vez generados los archivos ejecutables, se ha procedido a verificar que funcionan correctamente tanto en el simulador como en el propio procesador ARM, es decir,

en la placa S3CEV40. Para ello, se ha requerido configurar el debugger o depurador por medio de la instalación de un plugin denominado GDB.

Una vez configurado y comprobado que el debugger funciona sin problemas se da por concluida la preparación del entorno de trabajo sobre el que se van a realizar el resto de tareas que componen la práctica, descritas en los apartados siguientes.

3.Objetivos

A continuación se explican cuáles son los objetivos del trabajo realizado:

- Interactuar con una placa real, y ejecutar programas con ella y sobre el simulador
- Profundizar en la interacción entre lenguaje C y lenguaje ensamblador ARM.
- Depurar correctamente el código ensamblador que genera un compilador a partir de un lenguaje en alto nivel.
- Gestionar la entrada/salida con dispositivos básicos, asignando valores a los registros internos de la placa desde un programa en C
- Aprender a desarrollar en C las rutinas de tratamiento de interrupción.
- Aprender a utilizar periféricos reales, como los temporizadores internos de la placa.
- Conocer la estructura segmentada en tres etapas del procesador ARM , uno de los más utilizados actualmente en sistemas empotrados.
- Familiarizarse con el entorno Eclipse sobre Windows, con la generación cruzada de código para ARM y con su depuración.
- Desarrollar código en ensamblador ARM adecuado para optimizar el rendimiento.
- Optimizar código tanto en ensamblador ARM como utilizando las opciones del compilador optimizador, entendiendo en todo momento qué significa cada una de las posibles opciones.

- Entender la finalidad y el funcionamiento de las Aplication Binary Interface, ABI, en este caso el estándar AATPCS (ARM Application Procedure Call Standard), y combinar de manera eficiente código en ensamblador con código en C.

- Saber depurar código siguiendo el estado arquitectónico de la máquina: contenido de los registros y de la memoria.

- Gestionar el tiempo de trabajo del proyecto correctamente en función de la disponibilidad de acceso al sistema específico.

4.Metodología

4.1. Estudio de la documentación

Se comenzó por estudiar la documentación proporcionada para la realización de la práctica, continuando con la relacionada con la puesta en marcha del proyecto en Eclipse, tanto en simulador como en la placa real.

A continuación se revisaron las instrucciones disponibles en la arquitectura ARMv7 (*ARM quick reference card*) y el protocolo llamadas a funciones a seguir (A*ATPCS*).

4.2. Estudio y depuración de las fuentes

Una vez configurado Eclipse, se estudió el código fuente, principalmente el fichero reversi8_2018.c, para entender el funcionamiento del mismo.

También se realizaron diversas pruebas depurando el código. Estas se centraron en entender el código generado por el compilador en ensamblador, para comprender mejor el protocolo ATPCS y algunos detalles de la arquitectura y la configuración del proyecto, como por ejemplo hacia dónde se mueve la pila o cómo se realiza la indexación de una matriz.

4.3. Diseño e implementación de la función patron_volteo_arm_c

Tras entender el código generado por el compilador, se procedió a realizar un diseño de la función en pseudo-ensamblador. De esta manera se

localizaron las siguientes optimizaciones antes de implementar la función en ensamblador real.

4.3.1. Optimizaciones

Comparando con el código generado por el compilador, no se guardan los cuatro primeros parámetros de la función en la pila para desapilarse según se necesitan, ya que al tenerse directamente en registros se puede hacer uso de ellos sin pasar por memoria. De esta manera se consigue que solo haya que acceder a memoria cuando se va a llamar a una función o se esté trabajando con punteros. Sin embargo, al tomar esta decisión es necesario forzar al compilador a usar chars sin signo, ya que de la manera en que se están usando (para realizar operaciones aritméticas) se hace overflow y quedan bits en el registro que podrían generar problemas más adelante. Esto no sucede con el código generado por el compilador, porque al usar un parámetro, se carga de memoria un solo byte y se elimina el posible overflow de las operaciones anteriores. Al liberar o guardar espacio de la pila y no es necesario guardar variables en registros, se modifica el registro sp con instrucciones add/sub para evitar acceder a memoria.

También se mejoraron los saltos condicionales, por medio del uso de instrucciones predicadas y ejecutando las comparaciones pertinentes una sola vez, evitando así comparaciones innecesarias, ya que había lógica repetida en la expresión if del código en C original.

Por último, se intentó reducir el uso de registros, pudiendo usar el registro ip en vez del fp para tratar el direccionamiento de variables locales en pila.

4.3.2. Marco de pila

Al llamar a la función, y debido a que tiene siete parámetros, tres de ellos se pasan a través de la pila. Solo hay que guardar estas tres variables en registros, no desapilarlas, ya que de esto se encarga la función invocante.

El marco de pila, tras el bloque de activación queda así:

Posición respecto de sp al comienzo de la función	Registro o variable en una posición
+8	color
+4	SC
0	SF
-4	lr
-8	ip (sp al comienzo de la función)
-12	r9
-16	r8
-20	r7
-24	r6
-28	r5
-32	r4
-36	hueco para posicion_valida

Antes de llamar a ficha_valida, se guardan en la pila los registros correspondientes a los parámetros de la función, ya que son scratch y la función llamada no tiene que preservarlos.

Posición respecto de sp al comienzo de la función	Registro o variable en una posición
-40	r3 (CA)
-44	r2 (FA)
-48	r1 (longitud)
-52	r0 (tablero)

Tras ejecutar ficha_valida, se desapilan los registros scratch y también el hueco para posicion_valida, ya que la variable ha sido modificada por la función llamada y será necesaria en adelante, quedando sp en +32.

Antes de llamarse recursivamente, se guardan en la pila los 3 últimos parámetros de la función, liberándose su espacio tras la llamada sin acceder a memoria, por medio de un add.

Posición respecto de sp al comienzo de la función	Registro o variable en una posición
-36	color
-40	SC
-44	SF

Tras el bloque de desactivación, la pila queda así:

Posición respecto de sp al comienzo de la función	Registro o variable en una posición
+8	color
+4	SC
0	SF

4.4. Diseño e implementación de la función patron_volteo_arm_arm

Al terminar con la implementación de la función patron_volteo_arm_c, se procedió a implementar ficha_valida en ensamblador para realizar el inlining en la función anterior.

4.4.1. Optimizaciones

Al realizarse un inlining, se eliminan por completo todos los accesos a memoria necesarios para llamar a ficha_valida. Además, se vuelve a hacer uso de las instrucciones predicadas en ficha_valida, el indexado de la matriz tablero de reduce a dos instrucciones gracias al uso del barrel shifter, y se elimina la necesidad de guardar la variable local posicion_valida en la pila, haciéndose uso de un registro para almacenarla. El resto del código de la función queda igual que en la versión anterior.

4.4.2. Marco de pila

El marco de pila queda parecido al de la función anterior. Tras el bloque de activación queda así:

Posición respecto de sp al comienzo de la función	Registro o variable en una posición
+8	color
+4	SC
0	SF
-4	lr
-8	ip (sp al comienzo de la función)
-12	r9
-16	r8
-20	r7
-24	r6
-28	r5
-32	r4

Antes de llamarse recursivamente, se guardan en la pila los 3 últimos parámetros de la función, liberándose su espacio tras la llamada sin acceder a memoria, por medio de un add.

Posición respecto de sp al comienzo de la función	Registro o variable en una posición
-36	color
-40	SC
-44	SF

Tras el bloque de desactivación, la pila queda así:

Posición respecto de sp al comienzo de la función	Registro o variable en una posición
+8	color
+4	SC
0	SF

4.5. Verificación automática de las funciones realizadas

Se ha implementado la función patron_volteo_test, la cual llama a las funciones patron_volteo, patron_volteo_arm_c y patron_volteo_arm_arm. Si las llamadas a las últimas dos funciones no devuelven lo mismo que la original, la función queda en un bucle infinito, indicando que el resultado obtenido no es el correcto. Se invoca a esta función desde elegir_mov y actualizar_tablero.

Además, se ha implementado una simulación de un juego (función reversi8_test), la cual realiza varios movimientos comprobando en cada paso que las fichas se mueven en el tablero como deberían, verificando así el correcto funcionamiento del sistema con todas las funciones. Si los tableros resultantes al mover las piezas no son los correctos, la función queda en un bucle infinito.

4.6. Estudio de fuentes de timer0 y documentación sobre el mismo

Para poder entender el funcionamiento del timer y las interrupciones en el chip S3C44B0X se estudió el código proporcionado en los ficheros timer.h, timer.c y los temas 9 y 11 del documento sobre el chip.

4.7. Diseño e implementación de biblioteca para utilizar timer2

Tras entender el funcionamiento del timer y como configurarse se pasó a implementar la función timer2_inicializar, la cual configura las líneas de interrupción como IRQ, habilita en el vector de máscaras de interrupción el bit del timer2, establece la rutina del servicio para timer2 a la función timer2_ISR, la cual aumenta una variable estática propia del módulo timer2_num_int, establece en los registros apropiados el preescalado, divisor

e intervalos superior e inferior de la cuenta del timer para conseguir la máxima precisión posible e inicializa el timer2 con auto reload y update manual.

Se optó por definir varias constantes en un enum, para poder cambiarlas rápidamente si fuese necesario. Al final, para obtener la máxima precisión posible, se usó un preescalado de 1 (TIMER2_PRESCALER), el mayor intervalo de la cuenta posible para no sobrecargar al procesador (constantes TIMER2_INTERVALO_SUPERIOR y TIMER2_INTERVALO_INFERIOR) con interrupciones y un divisor de la frecuencia de ½ (TIMER2_DIVISOR). También se define una constante (TIMER2_PERIODO_US) que mide el tiempo en µs por cada tick del procesador, calculada de la siguiente manera:

1.0 / (PROCESADOR_FRECUENCIA_MHZ / (TIMER2_PRESCALER + 1) / TIMER2_DIVISOR)

La función timer2_empezar para el timer, resetea las timer2_num_int y la variable de cuenta del timer y vuelve a inicializar el timer.

La función timer2_leer hace uso de la constante definida anteriormente para calcular cuántos µs lleva contados el timer, por medio de la siguiente fórmula:
TIMER2_PERIODO_US * (timer2_num_int * (TIMER2_INTERVALO_SUPERIOR - TIMER2_INTERVALO_INFERIOR) + (TIMER2_INTERVALO_SUPERIOR - rTCNTO2))

La función timer2_parar para el timer y devuelve el resultado de llamar a timer2_leer.

4.8. Medidas de rendimiento y optimizaciones del compilador

Una vez verificado que el código funciona bien en la placa y que el timer2 mide los tiempos correctamente se ha procedido a analizar los tiempos de ejecución reales sobre el procesador de la placa. Para ello, solo se han tenido en cuenta las funciones críticas, no todo el programa completo, por lo que únicamente se han medido las funciones ficha_valida y patron_volteo(), patron_volteo_arm_c y patron_volteo_arm_arm.

Para medir el rendimiento se ha procedido a calcular el tamaño en bytes de cada una de las funciones anteriores teniendo en cuenta las diferentes optimizaciones del compilador. A continuación se encuentran tablas comparativas donde se muestran los tamaños de cada función dependiendo de los flags de optimización en bytes en el caso de las funciones en C y el tamaño en bytes fijo en caso de las funciones en

ensamblador, ya que al estar ya en ensamblador el compilador no las modifica.

Función analizada	Flag -00 (bytes)	Flag -01 (bytes)	Flag -02 (bytes)	Flag -03 (bytes)	Flag -0s (bytes)
ficha_valida	212	88	100	100	96
patron_volteo	284	208	204	1840	204

Función analizada	Bytes
patron_volteo_arm_c	112
patron_volteo_arm_arm	132

También se han medido los tiempos que tarda en ejecutarse cada función, realizando 10 medidas y tomando su media. A continuación se proporciona una tabla descriptiva de los resultados obtenidos.

Función	Flag -00 (µs)	Flag -01 (µs)	Flag -02 (µs)	Flag -03 (µs)	Flag -0s (µs)
patron_volteo	25	14	14	6	14
patron_volteo_arm_c	22	12	13	13	13
patron_volteo_arm_arm	12	8	8	9	8

Pese a que no debería, el tiempo de ejecución cambia según se usen o no flags de optimización para las funciones realizadas en ensamblador. Esto se debe probablemente a optimizaciones realizadas en el código ensamblador generado para llamar a las funciones, ya que una vez se usan flags de optimización no se aprecian cambios significativos en los resultados, y a que patron_volteo_arm_c todavía llama a la función ficha_valida, que sí será optimizada por el compilador.

5.Resultados y análisis

Con los resultados obtenidos, se puede observar una mejora en el tiempo de ejecución tras activar flags de optimización en las funciones en C, e incluso en las funciones en ensamblador, probablemente por una mejora en las instrucciones de llamada a función. La mejora es de alrededor de un factor de 2 para patron_volteo en todos los flags, excepto en O3, el cual realiza optimizaciones agresivas llegando a obtener una mejora de un factor de 4, aumentando sustancialmente el tamaño ocupado por la función. Esta optimización consiste en cambiar la función de recursiva a iterativa, evitando así acceder tanto a memoria al llamarse a sí misma.

Para elegir una función a usar en adelante entre las tres versiones, es necesario tener en cuenta el tamaño en bytes y el tiempo en ejecución de la función. En un primer análisis se puede descartar el uso de patron_volteo_arm_c, ya que es más lenta que patron_volteo_arm_arm en todos los casos y más lenta que patron_volteo con flag de optimización O3, y no existe una gran diferencia en tamaño con patron_volteo_arm_arm. Entre patron_volteo y patron_volteo_arm_arm, sería necesario evaluar si se cuenta con memoria suficiente para justificar ocupar 18 veces más memoria para ganar un poco en tiempo de ejecución en O3. En esta placa se cuenta con una memoria flash de 1Mx16bit y una memoria SDRAM de 4Mx16bit, por lo tanto para este proyecto se cuenta con memoria de sobra, y se opta por patron_volteo, con flag de optimización de O3.

Sin embargo, si no se contase con tanta memoria se optaría por la función patron_volteo_arm_arm, ya que no es significativamente más lenta y ocupa mucho menos.

6.Conclusiones

Esta práctica es muy interesante ya que es la primera vez que se ha tenido que programar usando dos lenguajes de programación distintos en un mismo entorno de trabajo. Paralelamente se ha tomado conciencia de lo importante que es realizar códigos de programas optimizados y de lo útil que es observar cómo los cambios que se efectúan sobre dichos códigos influyen de manera tan notoria.

También ha sido interesante conocer los diferentes niveles de optimización con los que cuenta el compilador de C, y de cómo afecta al rendimiento cuando se compila con cada uno de ellos. No obstante la conclusión más importante a la que se ha llegado es que en todo momento se debe saber lo que se está haciendo y que cualquier paso que se efectúe no debe realizarse a la ligera, sino que hay que tener en cuenta todos los conocimientos que uno posee y analizar con sumo cuidado las posibles consecuencias que puede tener cada paso que se da.

Durante la realización de la práctica se han tenido que afrontar varios problemas como por ejemplo a la hora de trabajar con enteros con signo y sin signo, signed y unsigned

respectivamente, se han tenido ciertos problemas ya que se gestionan diferente forma y al debuggear el código los contenidos de los registros cambiaban y producían resultados inesperados como ocurría en la función patron_volteo() a la hora de moverse en las direcciones del tablero. Para solucionar este error hubo que incorporar al compilador el flag -fsignedchar en la opción miscellaneous del compilador para que hiciera extensión automática del signo. Paralelamente hubo que modificar el código de algunas instrucciones para adaptar los comparadores de enteros sin signo a comparadores de enteros sin signo.

Otro problema muy importante fue la configuración del timer2, concretamente los registros porque fueron puestos en formato binario con el símbolo 0b para indicarle al compilador que es base binaria. Sin embargo al compilar y ejecutar los valores de los registros se almacenaban bien pero no se producían las interrupciones. Se estuvo mucho tiempo analizando los posibles problemas del error hasta que se optó por cambiar los contenidos a formato hexadecimal, fue entonces cuando finalmente se solucionó el problema.

Por último, también hubo que adaptar ligeramente el código para poder obtener los valores de tiempo al realizar mediciones, ya que al ejecutarse un código optimizado el compilador llegaba incluso a eliminar instrucciones si sus valores no eran usados posteriormente.

7. Actualizaciones

En esta entrega se han realizado dos importantes cambios en los códigos fuente entregados. Dichas modificaciones se han realizado en la configuración del timer2. En la versión inicial de la configuración de este periférico algunos registros de control eran gestionados de tal forma que su contenido era reasignado por completo. La razón por la que se utilizó este procedimiento fue porque en esta práctica solo se pedía trabajar con un único periférico, por lo que modificar todos los bits del registro no supone ningún problema.

Sin embargo, esta manera de proceder puede dar problemas si se quiere trabajar con un nuevo periférico ya que al asignar el valor al registro es muy probable que se altere el valor de algún bit por otro que no es el correcto, derivando así en problemas durante el tiempo de ejecución. Por lo que se ha optado por gestionar los contenidos de los registros mediante el uso de operaciones con puertas lógicas. De esta forma en cada paso sólo se modifican aquellos bits del registro que son necesarios y los restantes permanecen intactos.

Otro cambio que hubo que hacer fue cambiar la variable timer2_num_int de static a static volatile, dado que es una variable que se modifica desde una subrutina de interrupción, por lo que debe ser volatile. Si la variable timer2_num_int se deja sin volatile, puede ocurrir la situación de que al hacer uso de la variable se produzca una interrupción y no quede reflejado la actualización de la misma, puesto que el compilador puede optimizar la variable almacenándola en un registro.

8.Bibliografía

Como documentación de ayuda se ha empleado todo el material de apoyo proporcionado por el profesorado de la asignatura. Dicho material está compuesta por el manual del lenguaje ensamblador ARM, el estándar ATPCS, la documentación referente al chip S3C44B0X y la placa S3CEV40 y los archivos de la universidad complutense. Toda esta información se puede encontrar en la sección de material de prácticas de la asignatura Proyecto Hardware en Moodle.

9.Anexos

9.1. Código patron_volteo_arm_c

```
@ Parametros recibidos por registros
@ r0 = @tablero
@ r1 = @longitud
@ r2 = FA
@ r3 = CA
@ Parametros recibidos en pila y desapilados
@ r4 = SF
@ r5 = SC
@ r6 = color
@ Variables locales y longitud
@ r7 = posicion_valida
@ r8 = longitud
@ r9 = casilla (ficha_valida())

patron_volteo_arm_c:
          @ ip = @ donde estan las variables pasadas al
invocar la funcion
          mov ip, sp
          PUSH {r4-r9, ip, lr}
          sub sp, sp, #4 @ se guarda espacio para
posicion_valida
          ldmia ip, {r4, r5, r6} @ POP empezando en ip,
de las variables pasadas al invocar la funcion, IA ya que
          @ queremos coger la variable en SP al llamar a
patron_volteo y las 2 anteriores
```

```
            @ variables locales
            add r2, r2, r4   @ FA = FA + SF
            add r3, r3, r5   @ CA = CA + SC

            @ preparar llamada a ficha_valida
            PUSH {r0-r3}
            mov r1, r2 @ mover FA a r1
            mov r2, r3 @ mover CA a r2
            sub r3, ip, #36 @ *posicion_valida
            bl ficha_valida
            mov r9, r0 @ r9 = casilla (resultado de
ficha_valida())
            POP {r0-r3, r7} @ r7 = posicion_valida

            ldr r8, [r1] @ r8 = longitud
            cmp r7, #1 @ if posicion_valida != 1, salto a
else
            bne else
            cmp r9, r6 @ if casilla == color, se 'salta' a
elseif
if:         addne r8, r8, #1 @ r8++
            strne r8, [r1] @ guardar longitud en memoria
            @ preparar llamda a patron_volteo (recursiva)
            PUSHne {r4-r6} @ guardamos r4-r6 parametros
funcion
            blne patron_volteo_arm_c
            addne sp, sp, #12 @ 4 * 3 (subimos sp por el
PUSH anterior)
            bne fin @ en r0 tenemos ya el valor a devolver
elseif: cmp r8, #0
            movgt r0, #1
            bgt fin
else: mov r0, #0
fin:  POP {r4-r9, sp, pc}
```

9.2. Código patron_volteo_arm_arm

```
@ Parametros recibidos por registros
@ r0 = @tablero
@ r1 = @longitud
@ r2 = FA
@ r3 = CA
@ Parametros recibidos en pila y desapilados
@ r4 = SF
@ r5 = SC
@ r6 = color
```

```
@ Variables locales y longitud
@ r7 = posicion_valida
@ r8 = longitud
@ r9 = casilla (ficha_valida())

patron_volteo_arm_arm:
          @ ip = @ donde estan las variables pasadas al
invocar la funcion
          mov ip, sp
          PUSH {r4-r9, ip, lr}
          ldmia ip, {r4, r5, r6} @ POP empezando en ip,
de las variables pasadas al invocar la funcion, IA ya que
          @ queremos coger la variable en SP al llamar a
patron_volteo y las 2 anteriores

          @ variables locales
          add r2, r2, r4  @ FA = FA + SF
          add r3, r3, r5  @ CA = CA + SC

          @ llamada a ficha_valida

          @ r0 = *tablero, r2 = FA, r3 = CA, r7 =
posicion_valida
          @ if FA > 7 salta a else_pv
          cmp r2, #7
          @ if CA > 7 salta a else_pv
          cmple r3, #7
          bgt else_pv
          @ if FA < 0 salta a else_pv
          cmp r2, #0
          @ if CA < 0 salta a else_pv
          cmpge r3, #0
          blt else_pv
          @ se carga tablero[f][c] en r9 (casilla)
          add r9, r0, r2, LSL #3
          ldrb r9, [r9, r3]
          @ si tablero[f][c] == CASILLA_VACIA (0) salta
a else_pv (no se ejecutan las 2 siguientes instrucciones)
          cmp r9, #0
          @ r7 (posicion_valida) = 1
          movne r7, #1
          bne fin_pv
else_pv:@ r7 (posicion_valida) = 0
          mov r7, #0
          @ casilla = CASILLA_VACIA (0)
          mov r9, #0
```

```
fin_pv:    @ r9 es tablero[f][c] o CASILLA_VACIA
(variable casilla)

           ldr r8, [r1] @ r8 = longitud
           cmp r7, #1 @ if posicion_valida != 1, salto a
else
           bne else
           cmp r9, r6 @ if casilla == color, se 'salta' a
elseif
if:        addne r8, r8, #1 @ r8++
           strne r8, [r1] @ guardar longitud en memoria
           @ preparar llamda a patron_volteo (recursiva)
           PUSHne {r4-r6} @ guardamos r4-r6 parametros
funcion
           blne patron_volteo_arm_c
           addne sp, sp, #12 @ 4 * 3 (subimos sp por el
PUSH anterior)
           bne fin @ en r0 tenemos ya el valor a devolver
elseif: cmp r8, #0
           movgt r0, #1
           bgt fin
else: mov r0, #0
fin:  POP {r4-r9, sp, pc}
```