

A dark blue vertical bar on the left side of the page. A blue arrow points to the right from the bar, containing the date.

29-10-2021

Práctica 1

Desarrollo de código para el
procesador ARM

Several thin, curved lines in dark blue and light grey originate from the bottom left corner and curve upwards and to the right.

David Ros y Álvaro Fraidias

Contenido

1. Resumen ejecutivo	3
2. Introducción	3
3. Objetivos	3
4. Apartado A	3
4.1 Inicio y variables	3
4.2 Código en C.....	4
4.2.1 Tiempo ejecución	4
4.3 Código en ensamblador ARM.....	5
4.3.1 Tiempo ejecución	6
4.4 Comparación C y ARM.....	6
5. Parte B	7
5.1 Introducción	7
5.2 Bloque de activación	7
5.3 Pila	8
5.3 AAPCS	8
5.3.1 Reglas de uso.....	8
5.3.2 Reglas uso de la pila	9
5.4 Parte C	9
5.4.1 Cabecera e inicio	9
5.4.2 Candidatos_actualizar_C	10
5.4.3 Candidatos_propagar_c	11
5.4.3.1 Declaración variables e inicialización	11
5.4.3.2 Recorrer fila.....	11
5.4.3.3 Recorrer columna.....	11
5.4.3.4 Recorrer región	12
5.5 Parte ARM	13
5.5.1 Cabecera e inicio	13
5.5.2 Candidatos_actualizar_arm.....	14
5.5.3 Candidatos_propagar_arm	16
5.5.3.1 Declaración de variables e inicialización	16
5.5.3.2 Recorrer fila.....	17
5.5.3.3 Recorrer columna.....	17
5.5.3.4 Recorrer región	18
5.5.4 Candidatos_propagar_all	19

5.6 Medidas de rendimiento.....	21
6. Automatización de la solución	21
7. Errores	22
8. Conclusión	24
9. Bibliografía	24



**Reconocimiento-No comercial-Sin
derivados 4.0 Internacional
(CC BY-NC-ND 4.0)**

1. Resumen ejecutivo

En el siguiente pdf se define el famoso juego sudoku para más tarde realizar la función más crítica de dicho juego: actualizar el tablero cuando se coloca un número. Una vez definido el problema se realiza la solución implementada tanto en C como en ARM para luego ir intercambiando código de una arquitectura a otra midiendo el rendimiento.

2. Introducción

Esta práctica se divide en dos apartados cuyos esqueletos son proporcionados por el profesor y debemos hacer un previo estudio para entenderlos: El primero consiste en implementar tanto en C como en ARM un programa que copie 25 valores de una dirección de memoria a otra y el segundo consiste en optimizar el rendimiento del juego “Sudoku” acelerando las funciones computacionalmente más costosas (críticas). Más adelante explicaremos con más detalle cada apartado.

3. Objetivos

Los objetivos de esta práctica son:

- Instalación y configuración correcta del entorno de desarrollo “Eclipse”.
- Volver a recordar el lenguaje de programación C y ARM.
- Entender la finalidad y el funcionamiento del estándar ATPCS.
- Aprender a desarrollar y optimizar código en ensamblador ARM y C.
- Comprobar la interoperabilidad entre C y ARM.

4. Apartado A

En este apartado se desarrollará un código tanto en C como en ARM para trasladar un bloque de datos de una dirección de memoria a otra.

4.1 Inicio y variables

```
// ARM assembly code for the start of the program

LDR    sp,=STACK      /* set up stack pointer (r13) */

LDR    r0, =src //Cargamos en r0 la direccion origen de los datos
LDR    r1, =dst1 //Cargamos en r1 la direccion destino de los datos
BX     ARM_copy_10     //Llamamos a la funcion arm

//Volvemos a cargar las direcciones porque al llamar a una funcion se han borrado
LDR    r0, =src
LDR    r1, =dst2

# function __c_copy is in copy.c

LDR    r3, = __c_copy_10
MOV    lr, pc
BX     r3

stop:
B      stop           /* end of program */
```

Ilustración 1: Inicio del programa

Como se puede ver en la ilustración 1, cargamos las direcciones de memoria origen y destino para llamar a la función en ARM. Después volvemos a cargar la dirección origen y la otra dirección destino y llamamos a la función en C.

```

.data
.ltorg /* guarantees the alignment */
src:
    .long    1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25
dst1:
    .long    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
dst2:
    .long    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
.end
#           END

```

Ilustración 2: Datos utilizados en el programa

4.2 Código en C

```

//-----
// Function Name: __c_copy
// This function copies 25 words from one address (src) to another (dst)
// src: pointer to the source block
// dst: pointer to the destination block
//-----
void __c_copy_10(int *src, int *dst)
{
    int i;
    for(i = 0; i < 25; i++){
        *dst = *src;
        dst++;
        src++;
    }
}

```

Ilustración 3. Código parte A en c

En la ilustración 3 se representa el método necesario para mover un bloque de datos de una zona de memoria a otra. Dicho método recibe dos punteros como parámetros: “src” apunta al bloque donde inicialmente están los datos que se pretenden mover y “dst” apunta al bloque donde finalmente se van a guardar los datos. Mediante un bucle iterado 25 veces se iguala el puntero del bloque destino al puntero del bloque inicial. Acto seguido se incrementa cada puntero para acceder a la siguiente posición de memoria.

4.2.1 Tiempo ejecución

Para calcular el tiempo de ejecución hemos hecho un bucle que se ejecuta 1000 veces y hemos medido el tiempo manualmente.

Programa	Número de instrucciones	Tiempo de ejecución(milisegundos)
__c_copy_10	76	9,8

4.3 Código en ensamblador ARM

En la ilustración 4 se representa el código en ensamblador cuya funcionalidad es la misma que el de la ilustración 1. En este caso utilizamos un bucle for donde se iterará 4 veces en las cuales en cada iteración se copiarán 9 palabras. Gracias a LDM y STM podemos cargar o almacenar varias palabras simultáneamente.

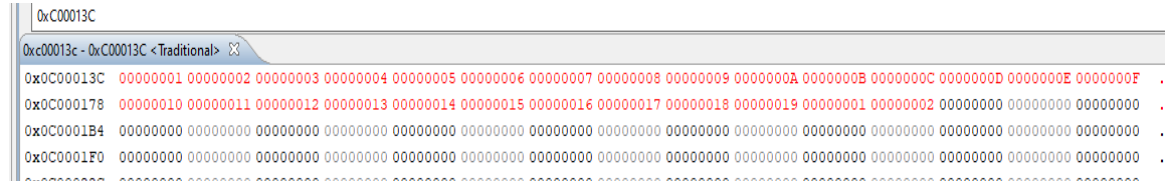
```
ARM_copy_10:
    push {r0-r11}
    MOV r2, #1

Loop:
    CMP r2, #4
    BGE Done
    LDM r0!, {r3-r11} //Carga 9 valores de la fuente
    STM r1!, {r3-r11} //Escribimos 9 valores en el destino
    ADD r2, r2, #1
    B Loop

Done:
    # restore the original registers
    pop {r0-r11}
    # return to the instruction that called the routine and to arm mode
    BX     r14
```

Ilustración 4. Código parte A en ensamblador ARM

Como las direcciones destino de ambos métodos son contiguas, en la última iteración de este código sobrescribe 2 valores en la dirección destino de C. Esto supondría un error si el método en C leyese de memoria ya que estaría leyendo un valor erróneo. Como escribe no pasa nada ya que sobrescribe el valor.



Address	Value
0xC00013C	00000001 00000002 00000003 00000004 00000005 00000006 00000007 00000008 00000009 0000000A 0000000B 0000000C 0000000D 0000000E 0000000F
0xC000178	00000010 00000011 00000012 00000013 00000014 00000015 00000016 00000017 00000018 00000019 00000001 00000002 00000000 00000000 00000000
0xC0001B4	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0xC0001F0	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Ilustración 5. Demostración de que en la última iteración copia 2 valores más de la cuenta

4.3.1 Tiempo ejecución

Igual que en el caso anterior, para calcular el tiempo de ejecución hemos hecho un bucle que se ejecuta 1000 veces y hemos medido el tiempo manualmente.

Al hacerlo nos hemos dado cuenta de que necesitamos dos registros:

- Uno para iterar el bucle
- Otro para comparar y saber cuándo se ha llegado al final del bucle.

Por consiguiente, debemos copiar menos palabras en cada iteración (exactamente 5 valores).

El código generado es el siguiente:

```
ARM_copy_10:
    push {r4-r8}
    MOV r2, #1
Loop:
    CMP r2, #6
    BGE Done
    LDM r0!, {r4-r8} //Carga 5 palabras de la fuente
    STM r1!, {r4-r8} //y ponerlos en el destino
    ADD r2, r2, #1
    B Loop
Done:
    # restore the original registers
    pop {r4-r8}
    # return to the instruction that called the routine and to arm mode
    BX    r14
```

Ilustración 6. Bucle ARM parte A

4.4 Comparación C y ARM

Se puede observar que el número de instrucciones y por lo tanto el tiempo de ejecución del programa en ARM tiene valores inferiores ya que en ARM utilizamos las instrucciones de carga y almacenamiento múltiple de forma que podemos cargar y almacenar 5 datos a la vez lo cual nos permite ejecutar el bucle sólo 5 veces. Sin embargo, el programa en C tiene un bucle que se ejecuta 25 veces y por lo tanto es menos eficiente.

$$\frac{76 \text{ instrucciones}}{34 \text{ instrucciones}} = 2,235 \text{ veces mejor el código en ARM}$$

5. Parte B

5.1 Introducción

En esta parte se pretende desarrollar la función crítica del juego Sudoku llamada “candidatos_actualizar” que como bien dice su nombre se encarga de recorrer el tablero comprobando si en cada celda hay un valor y en caso de haberlo, propagar ese valor para que las demás celdas lo eliminen de sus valores candidatos.

En caso de ser necesario propagar, se llamará desde “candidatos_actualizar” a otra función llamada “candidatos_propagar” que propagará ese valor por toda la fila, columna y región de la celda protagonista.

A la hora de realizar la práctica se ha diseñado estos dos métodos tanto en C como en ARM para acto seguido, combinar todas las posibilidades: C_C, C_ARM, ARM_C y ARM_ARM y evaluar sus rendimientos. También se ha hecho una función en ARM llamada “candidatos_arm_all” que combina dentro de ella la función “candidatos_actualizar_arm” y “candidatos_propagar_arm” y una función llamada “cuadrícula_candidatos_verificar” para verificar que el tablero resultante es correcto.

5.2 Bloque de activación

Salvo que la subrutina pueda realizar las tareas utilizando exclusivamente los registros destinados al paso de parámetros, será necesario crear y gestionar un espacio de memoria donde la subrutina pueda almacenar la información que necesita durante su ejecución. A este espacio de memoria se le denomina bloque de activación de la subrutina. El bloque de activación de una subrutina cumple los siguientes cometidos:

- En el caso que la subrutina llame a otras subrutinas, almacenar la dirección de retorno original.
- Proporcionar espacio para las variables locales de la subrutina.
- Almacenar los registros que la subrutina necesita modificar y que el programa que ha hecho la llamada no espera que sean modificados.
- Mantener los valores que se han pasado como argumentos a la subrutina.

5.3 Pila

Para implementar el bloque de activación necesitamos una estructura de datos conocida como pila. Para implementar dicha pila debemos reservar espacio en la memoria e indicarle al registro 13 (registro sp) la dirección de inicio de dicho espacio. En nuestro caso lo hacemos al inicio del programa ejecutando la siguiente instrucción:

```
MOV    sp, #0x4000 //set up stack pointer (r13)
```

Ilustración 7. Carga de la dirección pila en el r13

0101	r10	0
1010	r11	0
1010	r12	0
1010	sp	0x00004000

Ilustración 8. Ejecución de la ilustración 7

5.3 AAPCS

AAPCS son las siglas de “ARM Architecture Procedure Call Standard”. Este estándar implementa una serie de reglas que explicaremos en los siguientes subapartados.

5.3.1 Reglas de uso

- Los parámetros se pasan entre subrutinas a través de los registros R0-R3. La subrutina llamada no necesita restaurar el contenido de los registros R0 a R3 antes de regresar. En nuestro caso hemos almacenado los registros R0-R3 en la pila ya que dentro de las subrutinas llamamos a otras subrutinas por lo que necesitamos recuperar los valores. También hemos usado algunos de estos registros como registros auxiliares.
- En la subrutina, se usan los registros R4-R11 para guardar las variables locales. El valor de estos registros debe guardarse al inicio de la subrutina y restaurarse al final.
- El registro R12 se utiliza como registro temporal intermedio para la llamada de procedimiento y se indica como IP.
- El registro R13 se utiliza como puntero de pila y se denota como SP. El valor del registro SP al entrar en la subrutina y el valor al salir de la subrutina deben ser iguales.
- El registro R14 se denomina registro de conexión y se denota como LR. Se utiliza para guardar la dirección de retorno de la subrutina. Si la dirección de retorno se guarda en la subrutina, el registro R14 se puede utilizar para otros fines.
- El registro R15 es el contador del programa, denominado PC. No se puede utilizar para otros fines.

5.3.2 Reglas uso de la pila

ATPCS estipula que la pila es de tipo FD (Descendente completo donde sp apunta al último valor introducido y la pila de datos crece de la dirección alta a la dirección baja), es decir, la pila de disminución completa y la operación de la pila es una alineación de 8 bytes. Por tanto, los comandos de uso frecuente son STMFD y LDMFD.

En nuestro caso nos dimos cuenta que daba igual usar las instrucciones STDMD y LDMFD o las instrucciones PUSH y POP. Sin embargo, hemos usado las instrucciones STDMD y LDMFD para ceñirnos exactamente al estándar.

5.4 Parte C

5.4.1 Cabecera e inicio

En esta parte de implementación en C se ha tenido que añadir una función “main” la cual es llamada desde el inicio del programa ARM. En esta función “main” se llama a la función “sudoku9x9”.

```
int main(void)
{
    if (sudoku9x9(cuadrícula_C_C, cuadrícula_C_ARM, cuadrícula_ARM_C, cuadrícula_ARM_ARM, solución) == 0) {
        return 0;
    }
    return -1;
}
```

Ilustración 9. Función “main”

La función “sudoku9x9” tiene como parámetros todas las cuadrículas de las diferentes configuraciones, así como la cuadrícula solución. En esta función se declara un vector de celdas vacías con 5 posiciones, y se llama a cada una de las configuraciones para guardar el número de celdas vacías en cada una de las posiciones del vector.

```
int
sudoku9x9(CELDA cuadrícula_C_C[NUM_FILAS][NUM_COLUMNAS],
CELDA cuadrícula_C_ARM[NUM_FILAS][NUM_COLUMNAS],
CELDA cuadrícula_ARM_ARM[NUM_FILAS][NUM_COLUMNAS],
CELDA cuadrícula_ARM_C[NUM_FILAS][NUM_COLUMNAS],
CELDA solución[NUM_FILAS][NUM_COLUMNAS])
{
    int celdas_vacias[5];
    int cuadrículas_correctas[5];
    int correcto = 0;
    int error = 1;
    size_t i;

    //VERSION C_C
    celdas_vacias[0] = candidatos_actualizar_c_c(cuadrícula_C_C);
    //Version C_ARM
    celdas_vacias[1] = candidatos_actualizar_c_arm(cuadrícula_C_ARM);
    //VERSION ARM_C
    celdas_vacias[2] = candidatos_actualizar_arm_c(cuadrícula_ARM_C);
    //VERSION ARM_ARM
    celdas_vacias[3] = candidatos_actualizar_arm_arm(cuadrícula_ARM_ARM);
    //VERSION ARM ARM ALL
    celdas_vacias[4] = candidatos_actualizar_all(cuadrícula_ARM_ARM);
}
```

Ilustración 10. Función “sudoku9x9”

5.4.2 Candidatos_actualizar_C

De esta función existen dos versiones, una que llama a la función propagar implementada en C y otra que llama a la función propagar implementada en ARM. Sus implementaciones son idénticas salvo una llamada a una función función (candidatos_propagar) y por lo tanto sólo se va a explicar la implementación de “candidatos_actualizar_c_c”.

Dicha función, mostrada en la ilustración 11, realiza 2 bucles anidados, que recorren las filas y las columnas de la cuadrícula que se ha pasado a la función como parámetro. Una vez que se están recorriendo las filas y las columnas se almacena el valor de la celda correspondiente llamando a la función “celda_leer_valor”. Si el valor obtenido de esta función es 0 significa que no hay ningún valor en esa celda, y, por lo tanto, debemos actualizar el contador de casillas vacías. Si el valor obtenido es distinto de 0 quiere decir que existe un valor en dicha celda y debe ser propagado haciendo uso de la función “candidatos_propagar_c”.

Por último, esta función devuelve el número total de celdas vacías de la cuadrícula.

```
//EJECUTA ACTUALIZAR EN C Y PROPAGAR EN C
static int candidatos_actualizar_c_c(CELDA cuadrícula[NUM_FILAS][NUM_COLUMNAS])
{
    int celdas_vacias = 0;
    int i = 0;
    int j;
    uint8_t valor = 0;
    while(i<NUM_FILAS){
        j = 0;
        while(j<NUM_COLUMNAS){ //No se (j<NUM_COLUMNAS) porque hay más columnas que filas debido al padding
            valor = celda_leer_valor(cuadrícula[i][j]);
            if(valor == 0){
                //actualizar contador de celdas vacias
                celdas_vacias++;
            }else{ //Si es distinto de 0 es porque hay un valor y debe ser propagado
                candidatos_propagar_c(cuadrícula,i,j, valor);
            }
            j++;
        }
        i++;
    }
    //retornar el numero de celdas vacias
    return celdas_vacias;
}
```

Ilustración 11. Función “candidatos_actualizar_c_c”

5.4.3 Candidatos_propagar_c

Esta función se encarga de propagar el valor dado como parámetro por la fila, columna y región del tablero pasado también por parámetro.

Para explicar el funcionamiento debemos dividir el código en 3 partes:

5.4.3.1 Declaración variables e inicialización

```
void candidatos_propagar_c(CELDA cuadrícula[NUM_FILAS][NUM_COLUMNAS],
                           uint8_t fila, uint8_t columna, uint8_t valor )
{
    const uint8_t inicio_bloque[3]={0,3,6};
    uint8_t fila_inicio = 0;
    uint8_t columna_inicio = 0;
    uint8_t reset_columnas = 0; //Sirve para reiniciar el valor de la columna para volver a recorrer las columnas de la siguiente fila
    uint16_t fila_final_region = 0;
    uint16_t columna_final_region = 0;
```

Ilustración 12: Declaración variables "candidatos_propagar_c"

La única variable que puede resultar rara de entender a simple vista es "inicio_bloque" ya que es una estructura que nos sirve para establecer la fila y columna inicial de la región donde se encuentra el valor que queremos recorrer.

5.4.3.2 Recorrer fila

```
/* recorrer fila descartando valor de listas candidatos */
while(columna_inicio < NUM_FILAS){ //No se usa(j<NUM_COLUMNAS) porque hay más columnas que filas debido al padding
    celda_eliminar_candidato(&cuadrícula[fila][columna_inicio], valor);
    columna_inicio++;
}
```

Ilustración 13: Bucle para recorrer las filas del tablero

Mediante un bucle recorreremos las filas del tablero eliminando los candidatos.

5.4.3.3 Recorrer columna

```
/* recorrer columna descartando valor de listas candidatos */
while(fila_inicio < NUM_FILAS){ //No se usa(j<NUM_COLUMNAS) porque hay más columnas que filas debido al padding
    celda_eliminar_candidato(&cuadrícula[fila_inicio][columna], valor);
    fila_inicio++;
}
```

Ilustración 14: Bucle para recorrer las columnas del tablero.

Mediante un bucle recorreremos las columnas del tablero eliminando los candidatos. En este caso no usamos la variable NUM_COLUMNAS ya que hay más columnas que filas debido al padding. El padding consiste en poner 0 en las filas de la cuadrícula a la hora de guardarlo en la memoria para poder ver en memoria el tablero correctamente.

5.4.3.4 Recorrer región

Para recorrer la región primero debemos saber en qué región nos encontramos. Para entender cómo saber en qué región nos encontramos para recorrerla lo hemos explicado mediante el siguiente dibujo:

	0			3			6		
0	X			X			X		
3	X			X			X		
6	X			X			X		

Ilustración 15: Dibujo para saber en qué región nos encontramos

Para recorrer la región debemos calcular la primera posición de la región en la que estamos, para ello usamos la fila y la columna que nos pasan como parámetro y guardamos en la pila.

Comparamos la fila y la columna con la fila y columna de la primera celda de cada región para saber si es mayor, menor o igual. De esta manera obtenemos la primera celda de la región que tenemos que recorrer.

Esta solución es menos eficiente que alguna de las propuestas en clase porque volvemos a recorrer 6 celdas que ya han sido revisadas.

Una vez explicado como sabemos que región recorrer vamos al código:

```
/* recorrer region descartando valor de listas candidatos */
//Se averigua en que cuadrante nos encontramos
if(fila < 3){ //Fila se encuentra en el primer cuadrante
    fila_inicio = inicio_bloque[0];
} else if((fila >=3 ) &&(fila < 6)){ //Fila se encuentra en el segundo cuadrante
    fila_inicio = inicio_bloque[1];
} else{ //Fila se encuentra en el tercer cuadrante
    fila_inicio = inicio_bloque[2];
}

if(columna < 3){ //Columna se encuentra en el primer cuadrante
    columna_inicio = inicio_bloque[0];
} else if((columna >=3 ) &&(columna < 6)){ //Columna se encuentra en el segundo cuadrante
    columna_inicio = inicio_bloque[1];
} else{ //Columna se encuentra en el tercer cuadrante
    columna_inicio = inicio_bloque[2];
}

fila_final_region = fila_inicio + 3;
columna_final_region = columna_inicio + 3;
reset_columnas = columna_inicio;
```

Ilustración 16: Averiguamos en que región estamos

En la ilustración 16 calculamos el inicio de la región a calcular y el final de dicha región. Después de esto, recorreremos un doble bucle como se puede apreciar en la ilustración 17.

```
//Se recorre la region
while(fila_inicio < fila_final_region){ //Se recorren las filas
    while(columna_inicio < columna_final_region){ //Se recorren las columnas
        celda_eliminar_candidato(&cuadrricula[fila_inicio][columna_inicio], valor);
        columna_inicio ++;
    }
    columna_inicio = reset_columnas;
    fila_inicio++;
}
```

Ilustración 17. Función “candidatos_propagar_c”

5.5 Parte ARM

En esta parte simplemente se traduce el código de C del subapartado anterior.

5.5.1 Cabecera e inicio

Para poder utilizar las funciones definidas en ARM desde C hay que escribir en la cabecera del fichero de ARM la palabra reservada “global” junto con el nombre de la subrutina que deseamos llamar desde C. Gracias a esto, estas subrutinas pueden ser llamadas desde otro fichero.

```
# ENTRY
.global start
.global candidatos_propagar_arm
.global candidatos_actualizar_c_arm
.global candidatos_actualizar_all
.global candidatos_actualizar_arm_arm
.global candidatos_actualizar_arm_c
```

Ilustración 18. Declaración de las subrutinas para poder usarlas desde otro fichero

En cuanto al programa principal es simple: se carga la función de main en un registro, se guarda la dirección actual en el registro lr para retornar cuando haya acabado el programa y salta al main (función implementada en C).

Si nos damos cuenta el salto se realiza con la etiqueta “X” lo que significa que realizará en modo Thumb (16 bits).

```
.extern main
    ldr r5, =main
    mov lr,pc
    bx r5

stop:
    B      stop      /* end of program */
```

Ilustración 19. Inicio programa

5.5.2 Candidatos_actualizar_arm

Hay dos versiones de esta función:

Una que llama a la función propagar implementada en C y otra que llama a la función propagar implementada en ARM.

Las dos son iguales salvo la llamada a esta función, es decir, son iguales salvo una instrucción por lo que no tiene sentido explicar las dos por separado ya que hacen lo mismo. Por ello cogemos de ejemplo la subrutina “candidatos_actualizar_arm_c” para explicarla:

```
candidatos_actualizar_arm_c:
    STMFD sp!, {r0-r7,lr}
    //r0 = direccion tablero
    //r1,r2,r3 //Se usan como registros auxiliares y como parametros de entrada-salida
    mov r4, #0 //int celdas_vacias = 0;
    mov r5, #0 //int i = 0;
    mov r6, #0 //int j;
    mov r7, #9 //Tamaño tablero
    B testBucleActualizarGrande
bucleActualizarGrande:
    B testBucleActualizarPequenio
testBucleActualizarPequenio:
    bucleActualizarPequenio:
        //valor = celda_leer_valor(cuadrícula[i][j]);
        ldr r0, [sp] // cargamos en el registro 0 la dirección de inicio de la cuadrícula
        mov r1, r5, LSL #5 //i * 32
        mov r2, r6, LSL #1 //j * 2
        add r3, r1, r2 // DESPLAZAMIENTO
        ldr r0, [r0,r3] //Cargamos el valor en r0 para llamar a la función "celda_leer_valor"
        bl celda_leer_valor
        cmp r0, #0 //if(valor == 0)
        beq celdasVaciasARM //Saltamos al final del bucle para no llamar a propagar
        ldr r0, [sp] // cargamos en r0 la dirección de la cuadrícula
        mov r1, r5 // cargamos en r1 la variable i
        mov r2, r6 // cargamos en r2 la variable j
        bl candidatos_propagar_c //candidatos_propagar_c(cuadrícula,i,j);
    finalBucleARM:
        add r6, r6, #1 //j++
    testBucleActualizarPequenio:
        cmp r6, r7 //Comparamos columnas con el final de tablero
        bne bucleActualizarPequenio
        add r5, r5, #1 //i++
        mov r6, #0 //j = 0;
    testBucleActualizarGrande:
        cmp r5, r7 //Compara las filas con el final de tablero
        bne bucleActualizarGrande
        add sp, sp, #4 //No queremos restaurar de pila r0 porque queremos retornar el número de celdas, por eso avanzamos a la siguiente instrucción
        mov r0, r4 //retornamos numero celdas vacias
        LDMFD sp!, {r1-r7,lr}
        mov pc, lr
celdasVaciasARM:
    addeq r4, r4, #1 // celdas_vacias++;
    b finalBucleARM //Vamos al final del bucle para volver a iterar
```

Ilustración 20. Subrutina “candidatos_actualizar_arm_c” implementada en ARM.

Para explicarla bien se va a comentar paso por paso:

```
candidatos_actualizar_arm_c:
    STMFD sp!, {r0-r7,lr}
    //r0 = direccion tablero
    //r1,r2,r3 //Se usan como registros auxiliares y como parametros de entrada-salida
    mov r4, #0 //int celdas_vacias = 0;
    mov r5, #0 //int i = 0;
    mov r6, #0 //int j;
    mov r7, #9 //Tamaño tablero
```

Ilustración 21. Variables “candidatos_actualizar_arm_c”

Son necesarios un mínimo 7 registros (es la solución más óptima ya que lo hemos revisado detenidamente una serie de veces y no hemos podido hacer esta subrutina con menos registros).

Los registros apilados en la pila quedarían de la siguiente manera:

R0	(#0)
R1	(#4)
R2	(#8)
R3	(#12)
R4	(#16)
R5	(#20)
R6	(#24)
R7	(#28)
LR	(#32)

Ilustración 22: Pila con el bloque de activación de la subrutina "candidatos_actualizar_C" (sp apunta a r0)

Una vez explicada la creación de variables e inicialización pasamos a explicar el cuerpo de la subrutina:

```

B testBucleActualizarGrande
bucleActualizarGrande:
    B testBucleActualizarPequenio
    bucleActualizarPequenio:
        //valor = celda_leer_valor(cuadrícula[i][j]);
        ldr r0, [sp] // cargamos en el registro 0 la dirección de inicio de la cuadrícula
        mov r1, r5, LSL #5 //i * 32
        mov r2, r6, LSL #1 // j * 2
        add r3, r1, r2 // DESPLAZAMIENTO
        ldr r0, [r0,r3] //Cargamos el valor en r0 para llamar a la función "celda_leer_valor"
        bl celda_leer_valor
        cmp r0, #0 //if(valor == 0)
        beq celdasVaciasARM //Saltamos al final del bucle para no llamar a propagar
        ldr r0, [sp] // cargamos en r0 la dirección de la cuadrícula
        mov r1, r5 // cargamos en r1 la variable i
        mov r2, r6 // cargamos en r2 la variable j
        bl candidatos_propagar_c //candidatos_propagar_c(cuadrícula,i,j);
        finalBucleARM:
        add r6, r6, #1 //j++
    testBucleActualizarPequenio:
        cmp r6, r7 //Comparamos columnas con el final de tablero
        bne bucleActualizarPequenio
        add r5, r5, #1 //i++
        mov r6, #0 //j = 0;
    testBucleActualizarGrande:
        cmp r5, r7 //Compara las filas con el final de tablero
        bne bucleActualizarGrande
        add sp, sp, #4 //No queremos restaurar de pila r0 porque queremos retornar el número de celdas, por eso avanzamos a la siguiente instruccion
        mov r0, r4 //retornamos numero celdas vacias
        LDMFD sp!, {r1-r7,lr}
        mov pc, lr

```

Ilustración 23. Bucle "candidatos_actualizar_arm_c"

Para recorrer todo el tablero se necesita hacer un doble bucle (uno para las filas y dentro de él, uno para las columnas). Las condiciones de los dos bucles deben ser iguales ya que el tablero es un cuadrado que tiene las mismas filas que columnas.

La ejecución dentro del bucle es simple: se carga en el registro r0 la dirección de memoria de la celda que se quiere saber el valor (se carga en r0 para usarlo como parámetro al llamar a la función "celda_leer_valor") y se comprueba si hay valor. En caso de haber valor llama a la función propagar y en caso de no haber valor a la función "celdaVacía" para que sume en celdas vacías e itere.


```

celdasVaciasARM:
addeq r4, r4, #1 // celdas_vacias++;
b finalBucleARM //Vamos al final del bucle para volver a iterar

```

Ilustración 24. Subrutina “celdasVaciasARM”

Como se ha dicho anteriormente, esta función simplemente aumenta en 1 la cantidad de celdas vacías del tablero y pone en pc la dirección de retorno del final del bucle para que itere.

5.5.3 Candidatos_propagar_arm

Esta subrutina junto con “candidatos_propagar_all” son las subrutinas más largas del proyecto por lo que la dividiremos en 4 partes para comentarlas mejor: declaración de las variables, recorrer fila, recorrer columna y recorrer región.

5.5.3.1 Declaración de variables e inicialización

```

candidatos_propagar_arm: // Propaga los candidatos en ARM
    STMFD sp!, {r0-r9,lr}
    //r0 = cuadrícula
    //r1 = Fila
    //r2 = columna
    //r3 = valor
    //r4 = iterador columna
    //r5 = NUM_COLUMNAS/FILAS
    mov r4, #0
    mov r5, #9
    mov r1, r1, LSL #5 // Calculamos cual es la fila donde tenemos que propagar (Filas* 32)
    add r0, r0, r1 //Sumamos a la direccion inicial la fila donde vamos a empezar a propagar

```

Ilustración 25. Declaración de las variables que vamos a usar

Los registros r8 y r9 se explicarán más adelante cuando los usemos.

La pila quedaría de la siguiente manera:

R0	(#0)
R1	(#4)
R2	(#8)
R3	(#12)
R4	(#16)
R5	(#20)
R6	(#24)
R7	(#28)
R8	(#32)
R9	(#36)
LR	(#40)

Ilustración 26: Pila con el bloque de activación de la subrutina “candidatos_propagar_C” (sp apunta a r0)

5.5.3.2 Recorrer fila

```
//RECORRER FILA DESCARTANDO VALOR DE LISTAS CANDIDATOS
B testBucleColumnas
bucleColumnas:
    ldr r1, [sp, #12] //Cargamos el valor de celda_leer_valor(cuadrícula[fila][columna]) como segundo parametro de la funcion "celda_eliminar_candidato"
    bl celda_eliminar_candidato
    add r0, r0, #2 //Nos desplazamos por las columnas de la fila
    add r4, r4, #1 //columna_inicio++
testBucleColumnas:
    cmp r4, r5
    bne bucleColumnas
```

Ilustración 27. Recorrer fila subrutina "candidatos_actualizar_arm"

En esta parte del código se ejecuta un bucle para recorrer todas las filas. Dentro de él se carga en r1 el valor de la cuadrícula desde la pila (la dirección de la cuadrícula ya la tenemos en r0) y llamamos al método "celda_eliminar_candidato".

5.5.3.3 Recorrer columna

```
// RECORRER COLUMNA DESCARTANDO VALOR DE LISTAS CANDIDATOS
//r0 = direccion tablero
//r1 = fila
//r2 = columna
//r3 = registro auxiliar
//r4 = columna_inicio
//r5 = NUM_COLUMNAS/FILAS
mov r4, #0 //Reseteamos el iterador
ldr r0, [sp] //Cargamos en r0 la direccion inicial del tablero
ldr r2, [sp, #8] //Cargamos en r2 el número de columnas desde la pila
mov r3, r2, LSL #1 // Calculamos cual es la columna donde tenemos que propagar (columnas * 2)
add r0, r0, r3 //Sumamos a la direccion inicial la columna donde vamos a empezar a propagar
B testBucleFilas
bucleFilas:
    ldr r1, [sp, #12] //Cargamos el valor de celda_leer_valor(cuadrícula[fila][columna]) como segundo parametro de la funcion "celda_eliminar_candidato"
    bl celda_eliminar_candidato
    add r0, r0, #32 //Nos desplazamos por de la filas
    add r4, r4, #1 //columna_inicio++
testBucleFilas:
    cmp r4, r5
    bne bucleFilas
```

Ilustración 28. Recorrer columnas subrutina "candidatos_actualizar_arm"

Esta parte del código es prácticamente igual que la parte anterior salvo que ahora iteramos de 32 bits en 32 bits ya que nos desplazamos en las filas y no en las columnas.

5.5.3.4 Recorrer región

```
// RECORRER REGION DESCARTANDO VALOR DE LISTAS CANDIDATOS
//r0 = direccion tablero
//r1 = fila (i)
//r2 = columna (j)
//r3 = auxiliar multiplicacion
//r4 = fila inicio cuadrante
//r5 = columna inicio cuadrante
//r6 = fila final region
//r7 = columna final region
//r8 = auxiliar multiplicacion
//r9 = auxiliar suma

//Se averigua en que cuadrante nos encontramos
//CALCULAMOS LA FILA INICIAL DEL CUADRANTE
ldr r1, [sp, #4] //Cargamos en r1 la fila donde estamos
cmp r1, #3 //if(fila < 3)
movlt r4, #0 //less than

cmp r1, #6 //if (fila < 6)
cmplt r1, #3 //if (fila >=3 )
movge r4, #3

cmp r1, #6 //if(fila => 6)
movge r4, #6

//CALCULAMOS LA COLUMNA INICIAL DEL CUADRANTE
ldr r2, [sp, #8] //Cargamos en r2 la columna donde estamos
cmp r2, #3 //if(columna < 3)
movlt r5, #0 //less than

cmp r2, #6 //if (columna < 6)
cmplt r2, #3 //if (columna >=3 )
movge r5, #3

cmp r2, #6 //if(columna => 6)
movge r5, #6

add r6, r4, #3 //fila_final_region = fila_inicio + 3;
add r7, r5, #3 //columna_final_region = columna_inicio + 3;

STMFD sp!, {r5} //Guardamos en la pila la columna inicio para resetear
```

Ilustración 29. Calculamos la región a recorrer en "candidatos_propagar_arm"

Igual que en el caso de recorrer la región implementado en C, primero debemos calcular en que región nos encontramos. Una vez calculada la región, sumamos 3 tanto a las filas como a las columnas para marcar el final de región. También guardamos en la pila el valor de la columna inicial para resetearla cuando lleguemos al final del bucle anidado. Por consiguiente, la pila quedaría de la siguiente manera:

R5 (#0)
R0 (#4)
R1 (#8)
R2 (#12)
R3 (#16)
R4 (#20)
R5 (#24)
R6 (#28)
R7 (#32)
R8 (#36)
R9 (#40)
LR (#44)

Ilustración 30: Pila tras apilar el valor de columna_inicio (sp apunta a r5)

Una vez calculada la región a recorrer, ejecutamos el siguiente bucle con otro bucle anidado para recorrerla:

```
B testBucleFilasGrande
bucleFilasGrande:
    B testBucleColumnasPequeno
    bucleColumnasPequeno:
        ldr r1, [sp, #16] //Cargamos en r1 el valor
        ldr r0, [sp, #4] //Cargamos en r0 la direccion inicial del tablero
        mov r3, r4, LSL #5 //Multiplicamos por 32 el registro 7 (fila inicio cuadrante)
        mov r8, r5, LSL #1 //Multiplicamos por 2 el registro 8 (columna inicio cuadrante)
        add r3, r8, r3 //Sumamos la posición fila cuadrante + posición columna cuadrante
        add r0, r0, r3 //Direccion inicio region = Direccion inicio + (filas* 32) + (columnas * 2)
        bl celda_eliminar_candidato
        add r5, r5, #1 // columna_inicio ++
    testBucleColumnasPequeno:
        cmp r5, r7
        bne bucleColumnasPequeno
        add r4, r4, #1 //fila_inicio++
        ldr r5, [sp] // Cargamos de la pila columna inicio para resetearla
    testBucleFilasGrande:
        cmp r4, r6 //Compara las filas con el final de la region
        bne bucleFilasGrande
        add sp, sp, #4 //Sumamos posicion al puntero de pila para "eliminar" el valor que hemos metido en la pila durante la ejecucion de esta subrutina
        LDMFD sp!, {r0-r9, lr}
        mov pc, lr
```

Ilustración 31. Bucle para recorrer la región "candidatos_propagar_arm"

En él hacemos lo mismo que en los bucles anteriores, guardamos en r1 el valor que queremos eliminar de los candidatos de la región, después calculamos la dirección de memoria y la guardamos en r0 y, por último, llamamos a “celda_eliminar_candidato”.

5.5.4 Candidatos_propagar_all

La diferencia entre “candidatos_propagar_arm” y “candidatos_propagar_all” es que en esta última se ejecuta “candidatos_actualizar_arm” y “candidatos_propagar_arm” en la misma subrutina por lo que no necesitamos guardar dos veces en la pila el bloque de activación. Esto hace que el programa sea un poco más optimo. Por el contrario, debemos usar prácticamente todos los registros.

Como sería tontería volver a poner todo el código ya que hace exactamente lo mismo y está comentado línea por línea en el proyecto, hemos pensado que sería mejor comentar las pequeñas diferencias entre ambos:

```
candidatos_actualizar_all:
    STMFD sp!, {r0-r11,lr}
    //-----
```

Ilustración 32. Bloque de activación "candidatos_propagar_all"

La pila quedaría de la siguiente manera:

R0 (#0)
R1 (#4)
R2 (#8)
R3 (#12)
R4 (#16)
R5 (#20)
R6 (#24)
R7 (#28)
R8 (#32)
R9 (#36)
R10 (#40)
R11 (#44)
LR (#48)

Ilustración 33: Bloque de activación "candidatos_propagar_all" guardado en la pila (sp apunta a r0)

Al recorrer las filas, columnas y región nos damos cuenta de que ya no necesitamos cargar de la pila las filas y las columnas ya que siempre hemos tenido ese valor en los registros r5 y r6. Por consiguiente, podemos eliminar las siguientes instrucciones:

```
//ldr r1, [sp, #8] //Cargamos en r1 la fila inicial (Podemos eliminar esta instruccion)
```

Ilustración 34. Instrucción eliminada en recorrer fila

```
//ldr r2, [sp, #12] //Cargamos en r2 el número de columnas desde la pila (Podemos ahorrar esta instruccion)
```

Ilustración 35. Instrucción eliminada en recorrer columna

```
//Se averigua en que cuadrante nos encontramos
//CALCULAMOS LA FILA INICIAL DEL CUADRANTE
//ldr r1, [sp, #8] //Cargamos en r1 la fila donde estamos

//CALCULAMOS LA COLUMNA INICIAL DEL CUADRANTE
//ldr r2, [sp, #12] //Cargamos en r2 la columna donde estamos
```

Ilustración 36. Instrucciones eliminadas en recorrer región

Sin embargo, hemos tenido que utilizar más registros a la hora de recorrer la región ya que los registros que en el método "candidatos_propagar_arm" guardaban los límites de dicha región están utilizados por la parte de código que simula "candidatos_actualizar_arm". Estos registros eran r6 y r7 y los hemos tenido que cambiar por r10 y r11:

```
add r6, r4, #3 //fila_final_region = fila_inicio + 3;
add r7, r5, #3 //columna_final_region = columna_inicio + 3;
```

Ilustración 37. Candidatos_propagar_arm

```
add r10, r8, #3 //fila_final_region = fila_inicio + 3;
add r11, r9, #3 //columna_final_region = columna_inicio + 3;
```

Ilustración 38. Candidatos_propagar_all

Por último, recalcar que debemos devolver el número de celdas vacías. Por ello desapilamos el valor de r0 que hemos puesto en la pila, cargamos el valor de “celdas_vacias” en r0 (valor almacenado en r4) y hacemos el pop de los demás registros de la pila para reestablecer valores y volver a la instrucción que llamó a esta subrutina.

```
LDMFD sp!, {r0} //Quitamos el valor inicial de r0 guardado en la pila al
//          inicio de la subrutina ya que tiene que devolver el número de celdas vacías
mov r0, r4 //retornamos numero celdas vacías
LDMFD sp!, {r1-r11,lr}
mov pc, lr
```

Ilustración 39. Final subrutina "candidatos_propagar_all"

5.6 Medidas de rendimiento

Configuración	Número de instrucciones	Tiempo (milisegundos)
C_C	43.808	4,46
C_ARM	33.037	3,47
ARM_C	42.789	4,4
ARM_ARM	82.529	8,03
ARM_ALL	32.130	3,45

Como podemos observar, salvo por la configuración ARM-ARM, el código implementado en ARM es mucho más eficiente en cuanto a número de instrucciones y de tiempo de ejecución.

6. Automatización de la solución

Para poder verificar que las 5 configuraciones realizadas generan la misma salida se ha llevado a cabo la función “cuadrícula_candidatos_verificar”, mostrada en la ilustración 40. Dicha función compara las diferentes configuraciones con la cuadrícula solución, y devuelve valor 0 si las cuadrículas son iguales y valor 1 en el caso contrario.

```
//COMPRUEBA SI EL TABLERO SE HA HECHO CORRECTAMENTE
static int
cuadrícula_candidatos_verificar(CELDA cuadrícula[NUM_FILAS][NUM_COLUMNAS],
                                CELDA solución[NUM_FILAS][NUM_COLUMNAS])
{
    uint8_t i;
    uint8_t j;
    int correcto = 0;
    int incorrecto = 1;
    while(i<NUM_FILAS){
        while(j<NUM_COLUMNAS){
            if(cuadrícula[i][j]!= solución[i][j]){
                return incorrecto; //Si hay una celda que no sea igual a la solución nos dará error
            }
            j++;
        }
        i++;
    }
    return correcto;
}
```

Ilustración 40. Función "cuadrícula_candidatos_verificar"

En el método sudoku9x9 ejecutamos todas las variantes y para verificar que se ejecutan correctamente llamamos a la función de la ilustración 41 pasando la cuadrícula resultante de cada variante y la cuadrícula solución (esta cuadrícula viene dada por el profesor). Guardamos el resultado de esa verificación en una estructura que después recorreremos para verificar que todas las cuadrículas son correctas.

```
//Verificamos que se ejecuta correctamente
cuadriculas_correctas[0] = cuadrícula_candidatos_verificar(cuadrícula_C_C,solucion);
cuadriculas_correctas[1] = cuadrícula_candidatos_verificar(cuadrícula_C_ARM,solucion);
cuadriculas_correctas[2] = cuadrícula_candidatos_verificar(cuadrícula_ARM_C,solucion);
cuadriculas_correctas[3] = cuadrícula_candidatos_verificar(cuadrícula_ARM_ARM,solucion);
cuadriculas_correctas[4] = cuadrícula_candidatos_verificar(cuadrícula_ARM_ARM,solucion);
for (i=1; i < 5; ++i) {
    if (cuadriculas_correctas[i] == 1) {
        return error;
    }
}
return correcto;
```

Ilustración 41: Automatización de la solución

7. Errores

1. Función propagar

Cuando revisamos la práctica nos dimos cuenta de que leemos el valor de la celda tanto en actualizar como en propagar por lo que vimos que era ineficiente y se podría optimizar. Por consiguiente, decidimos cambiar el esqueleto del código que nos da el profesor para añadir el parámetro “valor” en las llamadas de propagar tanto en C como en ARM:

```
void
candidatos_propagar_c(CELDAA cuadrícula[NUM_FILAS][NUM_COLUMNAS],
                      uint8_t fila, uint8_t columna, uint8_t valor );
void
candidatos_propagar_arm(CELDAA cuadrícula[NUM_FILAS][NUM_COLUMNAS],
                       uint8_t fila, uint8_t columna, uint8_t valor );
```

Ilustración 42. Declaración métodos propagar en “sudoku_2021.h”

De esta forma obtenemos el valor y no necesitamos hacer las siguientes instrucciones al llamar a la subrutina “candidatos_propagar_arm”.

```
candidatos_propagar_arm: // Propaga los candidatos en ARM
    STMFD sp!, {r0-r9,lr}
    //r0 = cuadrícula
    //r1 = Fila
    //r2 = columna
    //Tanto r3 como r4 se usan como pasos intermedios para obtener el valor de una celda dada la fila y la columna
    //-----
    // uint8_t valor = celda_leer_valor(cuadrícula[fila][columna]); Y LO GUARDAMOS EN LA PILA
    mov r3, r1, LSL #5 //FILAS * 32
    mov r4, r2, LSL #1 // COLUMNAS * 2
    add r4, r3, r4 // DESPLAZAMIENTO
    ldr r0, [r0,r4] //Cargamos el valor en r0 para llamar a la función "celda_leer_valor"
    bl celda_leer_valor
    STMFD sp!, {r0} //Guardamos en la pila el valor devuelto de la función "celda_leer_valor"
    ..
```

Ilustración 43. Inicio “candidatos_propagar_arm” antes de poner el cuarto parámetro

De esta forma nos ahorramos 7 instrucciones cada vez que llamamos a propagar.

En el caso de C nos eliminamos la siguiente instrucción cada vez que lo ejecutamos:

```
void candidatos_propagar_c(CELDA cuadrícula[NUM_FILAS][NUM_COLUMNAS],
                          uint8_t fila, uint8_t columna)
{
    const uint8_t inicio_bloque[3]={0,3,6};
    uint8_t fila_inicio = 0;
    uint8_t columna_inicio = 0;
    uint8_t reset_columnas = 0; //Sirve para reiniciar el valor de la c
    uint16_t fila_final_region = 0;
    uint16_t columna_final_region = 0;

    /* valor que se propaga */
    uint8_t valor = celda_leer_valor(cuadrícula[fila][columna]);
}
```

Ilustración 44. Inicio “candidatos_propagar_c” antes de poner el cuarto parámetro

2. Main

A la hora de cargar la dirección de cualquier cuadrícula desde ARM nos salía un error ya que dichas cuadrículas estaban definidas en el fichero “tableros.h” y eran estáticas. Para solucionar este problema creamos un método llamado main que será llamado desde ARM y dentro de él iniciamos el programa.

La llamada quedará de la siguiente manera:

```
.extern main
ldr r5, =main
mov lr,pc
bx r5
```

Ilustración 45. Inicio programa ARM

```
int main(void)
{
    if (sudoku9x9(cuadrícula_C_C, cuadrícula_C_ARM, cuadrícula_ARM_C, cuadrícula_ARM_ARM, solución) == 0) {
        return 0;
    }
    return -1;
}
```

Ilustración 46. Método main del fichero “sudoku_2021.c”

3. Static

Mediante avanzábamos con la práctica nos dimos cuenta de que los métodos definidos en un fichero diferente a donde los queríamos llamar y que estaban definidos bajo la etiqueta “static” solo podían llamarse desde ese mismo fichero. Esto nos impedía poder llamar a los métodos “celda_leer_valor” y “celda_eliminar_candidato” por lo que tuvimos que eliminar dicha etiqueta.

4. Global y extern

Al crear un método, tanto en ARM como en C, no podía ser llamado desde el otro lenguaje. Si hacíamos un método en ARM y queríamos ejecutarlo desde C debíamos poner en la cabecera del fichero de ARM la etiqueta “.global” junto con el nombre de la subrutina que deseábamos exportar.

Esto mismo ocurría con los métodos definidos en C que queríamos utilizar en ARM. En este caso debíamos poner la etiqueta “.extern” junto con el nombre del método que queríamos usar en el fichero ARM.

8. Conclusión

Esta práctica permite recapacitar sobre la eficacia de un código implementado en ARM frente a un código implementado en C a la vez que nos familiariza con el entorno Eclipse y la interoperabilidad de C-ARM.

9. Bibliografía

- 1º Apuntes de la asignatura Arquitectura y organización de computadores I
- 2º Apuntes de la asignatura Arquitectura y organización de computadores II
- 3º Prentice Hall.(1996). *ARM architecture reference manual*