
MEMORIA PRÁCTICA 1

PROYECTO HARDWARE

Grado en Ingeniería Informática

Universidad de Zaragoza

Curso 2024/2025

Autores: Nekane Díaz Montoya (870795) y Jorge Hernández Aznar (872838)

Grupo 431

ÍNDICE

INTRODUCCIÓN.....	2
RESUMEN.....	2
OBJETIVOS.....	2
RESULTADOS.....	3
Optimizaciones realizadas en el código de matriz3x3_operar_ARM().....	3
Mapa de Memoria.....	5
Marco de Activación de la Pila.....	6
Diferencias entre el marco de activación dado en AOC1 y en PH.....	7
Análisis temporal.....	8
Análisis espacial (comparación tamaño de instrucciones Thumb con ARM).....	9
CONCLUSIONES.....	11
ANEXOS.....	12
Anexo main.c.....	12
Anexo matriz 3x3_operar_ARM_C.s.....	15
Anexo matriz 3x3_operar_ARM.s.....	17
Anexo matriz3x3_operar_THB.....	18

INTRODUCCIÓN

En esta memoria describiremos los pasos seguidos, los objetivos cumplidos y los resultados obtenidos después de haber realizado la práctica 1 de la asignatura *Proyecto Hardware*.

Se ha realizado la práctica en el entorno Keil μ Vision, un entorno de desarrollo integrado (IDE) para la programación de microcontroladores basados en la arquitectura ARM.

Trabajaremos sobre el microcontrolador LPC2105, un microcontrolador de 32 bits basado en la arquitectura ARM7TDMI-S de NXP Semiconductors. Soporta tanto instrucciones Thumb (16 bits) como ARM (32 bits), lo que mejora la eficiencia en términos de tamaño de código y rendimiento.

RESUMEN

El problema al que nos enfrentamos en esta práctica se trata de una operación con matrices: $\text{Resultado}[N][N] = A[N][N] * B[N][N] + (C[N][N] * D[N][N])^T$ Con $N = 3$ mediante el cual deberemos obtener el número de términos no cero que se obtienen en la matriz Resultado de diferentes maneras llamadas desde una misma función (`matrizNxN_verificar`): mediante el código en C (`terminos_no_cero_C`), el código en ARM combinado con C mediante llamadas (`terminos_no_cero_ARM_C`), el código puramente en ARM (`terminos_no_cero_ARM`) y el código en THUMB (`terminos_no_cero_THUMB`) para finalmente comparar los resultados y comprobar mediante varias pruebas que el resultado de todas las formas diferentes es el mismo y es el correcto.

Además deberemos observar y medir el rendimiento de las diferentes formas en las que el código está hecho y a su vez cómo se comporta el compilador en las diferentes formas (-O0, -O1, -O2, -O3) y las diferencias en el código en ensamblador generado.

OBJETIVOS

Optimizar el rendimiento acelerando las funciones, familiarizarse con el entorno de trabajo o Entender funcionalidad del código suministrado, combinar código en ensamblador con código en C, conocer la finalidad y el funcionamiento del ABI (ATPCS), saber depurar el estado arquitectónico (registros y memoria), analizar y medir rendimiento, optimizar código en distintos lenguajes y verificar resultado y documentar los resultados.

RESULTADOS

Optimizaciones realizadas en el código de `matriz3x3_operar_ARM()`

Para llevar a cabo una optimización del código de la función en ARM se han seguido los siguientes pasos. Antes de comenzar se debe comentar que no se ha seguido una implementación por módulos ni funciones auxiliares, de este modo se evita sobrecargar con las operaciones de epílogo y prólogo de las funciones.

(1)

Primeramente se ha intentado reducir los saltos entre los bucles, es decir, el compilador en el punto básico de optimización `-O0`, produce dos saltos con esta estructura:

```
ini_for      CMP     RX, #X
             BGE     fin_for

             ; CUERPO DEL BUCLE

             ADD     RX, RX, #1
             B       ini_for
fin_for
```

Para ello hemos decidido poner la comparación al final, (COMPARACIÓN QUE NO TIENE POR QUÉ SER OBLIGATORIA) y suponiendo que inicialmente siempre entra en el bucle.

```
ini_for      ; CUERPO DEL BUCLE

             ADD     RX, RX, #1
             CMP     RX, #X
             BGE     ini_for
```

(2)

Seguidamente, para las funciones `multiplicar(A,B,Resultado)` y `multiplicar(D,C,E)` se aplicaron optimizaciones similares. En la versión del compilador de C básica, se utilizan `MUL` y `ADD`, y esta operación se puede realizar en una sola como es `MLA`, por ejemplo el siguiente código para acceder a una componente de `A[][]` era de esta forma previamente,

```
; R10 = #12, R6 = i, R8 = k, R0 = @A
MUL    R9, R10, R6
ADD    R9, R9, R8, LSL #2
LDR    R9, [R9, R0]
```

Se puede reducir combinando el `MUL` y `ADD` de esta forma,

```
MLA    R9, R10, R6, R0
LDR    R9, [R9, R8, LSL #2]
```

(3)

Asimismo otra optimización que se ha realizado ha sido los accesos a memoria. Para guardar los resultados en `Resultado[i][j]` y en `E[i][j]` no hace falta cargar el valor de estos en cada iteración y guardarlo para volver a cargarlo en la siguiente. De este modo se almacena el valor en un registro, en estos casos en R5, y se almacena al final del bucle. Por ende se hacen simplemente 9 stores en memoria, 1 store cada vez que termina el bucle anidado más arriba. Antes por cada bucle se realizaban 3 loads y 3 stores respectivamente. Por lo cual la reducción de accesos a memoria es considerable.

(4)

A continuación, se ha optado por realizar la transposición de la matriz E dentro de la segunda función `multiplicar(C,D,E)`. Ha sido simplemente en vez de guardar cada valor en `E[i][j]` se almacena en `E[j][i]` y así se evita tener que realizar otro bucle para transponer todos los elementos.

(5)

Mientras que se realiza la suma de matrices `Resultado` y `E`, se realiza la comparación para obtener los valores diferentes de cero. Para no utilizar una estructura `if` con saltos se ha optado por realizar la comparación e introducir obligatoriamente una instrucción con ejecución condicional como es `SUBEQ R0, R0, #1` siendo `r0 terminos_no_cero` inicialmente declarado como 9.

(6)

Cabe destacar que como para `Resultado[][]` y para `E[][]` (matriz auxiliar para guardar $C \times D$ inicialmente) se hacen simplemente stores en memoria no hace falta inicializarlo al comienzo de la función ya que se pierden los valores previos.

Del mismo modo no hace falta declarar la variable auxiliar `F[][]` donde guardábamos la matriz `E[][]` transpuesta ya que nos dimos cuenta de que se podían comprimir y reducir los bucles y realizar en un solo bucle `multiplicar(A,B,Resultado)` y `multiplicar(D,C,E)`, guardando directamente la suma en `Resultado[i][j]`.

Sin embargo era necesario calcular también la transpuesta de $D * C$ directamente de modo que decidimos usar las propiedades de las matrices $C^t D^t = (CD)^t$. Para ello en el bucle único calculamos el resultado de esta forma:

$$Resultado[i][j] = \sum_{k=0}^N A[i][k] B[k][j] + \sum_{k=0}^N C[j][k] D[k][i]$$

(7)

Además se calculó el número de terminos_no_cero al introducir el resultado en memoria. De esta manera se realiza todo en el mismo bucle ahorrándonos sobrecarga de bucles innecesarios.

Otra opción que se ha valorado para optimizar el código de ARM ha sido desenrollar el bucle total y por ello calcular las direcciones de acceso de forma manual, haciendo así un código sin saltos ni cálculos intermedios de las direcciones de acceso a memoria efectivos. A pesar de poder obtener tiempos de ejecución más bajos, esta opción se ha descartado debido a poca legibilidad, al gran aumento en tamaño de código y a la escasa portabilidad del código. Por ejemplo, si se aumenta la N, habría que reescribir de nuevo parte del programa.

Finalmente es importante remarcar que el diseño de ARM monolítico, con compresión de bucles no es comparable con el diseño inicial con funciones en C, para ello se ha creado la versión análoga a ARM en C de modo que en el análisis temporal se pueda comparar de forma más realista y precisa.

Mapa de Memoria

Inicialmente podemos observar que nuestras variables globales Test_A y Test_B se almacenan al comienzo de la memoria RAM que inicia en la dirección 0x40000000 como variables volátiles (le indica al compilador que la variable puede ser modificada en cualquier momento fuera del flujo natural de ejecución). Posteriormente le siguen el resto de variables locales del main.

La pila se almacena en posiciones altas en memoria (0x40003FFF) creciendo hacia posiciones más bajas.

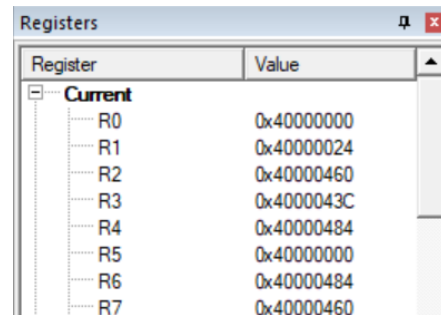
Conforme se hacen las llamadas a las funciones se va añadiendo espacio para formar la pila de activación, tras volver de estas se deshacen. Esto se explica más a fondo en el Marco de Activación de la Pila.

Test_A	0x40000000	32 KBYTE ON-CHIP STATIC RAM
Test_B	0x40000024	
...		
Instrucciones	0x40000420...0x40000464	128 KBYTE ON_CHIP FLASH MEMORY
Datos		
Test_C	0x40000460	
Test_D	0x4000043C	
Resultado	0x40000484	

Marco de Activación de la Pila

Inicialmente para llamar a la función **matrizNxN_operar_C** se pasa los cuatro primeros parámetros mediante los registros r0 a r3. No obstante el quinto hay que apilarlo en memoria de modo que se pueda acceder a él desde la función sin modificar los registros anteriores.

- r0 => @Test_A
- r1 => @Test_B
- r2 => @Test_C
- r3 => @Test_D
- SP = 0x40000438 => @Resultado(0x40000484)



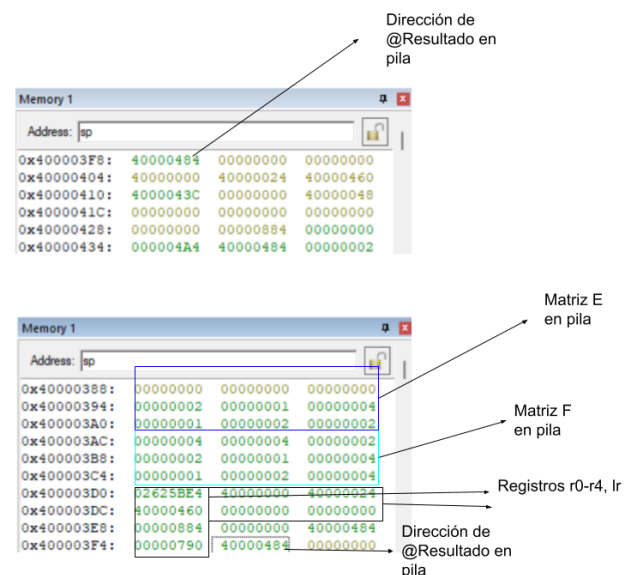
Register	Value
Current	
R0	0x40000000
R1	0x40000024
R2	0x40000460
R3	0x4000043C
R4	0x40000484
R5	0x40000000
R6	0x40000484
R7	0x40000460

Seguidamente se realiza un **STMDB** (Store múltiple con decremento anterior) para modificar el Stack Pointer (SP=r13) y apilar el Link Register (LR=r14) para poder obtener el PC de regreso y a continuación apila del registro r4 al registro r12.

A continuación de entrar en la subrutina/función se mueven los valores de los registros r0 al r3 a nuevos registros. Y en nuestro caso inicialmente se restan al SP el hueco para 18 enteros siendo cada mitad de estos el espacio reservado para las variables locales de la función F[N][N] y E[N][N] (en nuestra función operar sin optimizar, en la optimizada no es necesario reservar ese espacio para las matrices).

En la segunda llamada a una función **matrizNxN_multiplicar_C** se tiene que anidar la llamada de nuevo usando la pila y los registros de forma similar a la primera pero esta vez todos los parámetros tienen que pasarse en los registros.

- r0 => Test_A (0x40000000)
- r1 => Test_B (0x40000024)
- r2 => Resultado (0x40000484)



Dirección de @Resultado en pila

Matriz E en pila

Matriz F en pila

Registros r0-r4, lr

Dirección de @Resultado en pila

Address	Value
0x400003F8	00000000 00000000 00000000
0x40000404	00000000 00000000 00000000
0x40000410	00000000 00000000 00000000
0x4000041C	00000000 00000000 00000000
0x40000428	00000000 00000000 00000000
0x40000434	00000000 00000000 00000000
0x40000388	00000000 00000000 00000000
0x40000394	00000000 00000000 00000000
0x400003A0	00000000 00000000 00000000
0x400003AC	00000000 00000000 00000000
0x400003B8	00000000 00000000 00000000
0x400003C4	00000000 00000000 00000000
0x400003D0	00000000 00000000 00000000
0x400003DC	00000000 00000000 00000000
0x400003E8	00000000 00000000 00000000
0x400003F4	00000000 00000000 00000000

A continuación se produce otro **STMDB** para modificar el SP y almacenar de los registros r4 a r7. Esta vez no se almacena el LR porque no va a haber más llamadas a funciones anidadas dentro de esta.

Tras realizar la multiplicación se realiza el desapilamiento de los registros r4 a r7 con la operación **LDMIA**(Load Múltiple Con Incremento posterior) para devolver de nuevo el SP a la posición de la pila original antes de invocar a **matrizNxN_multiplicar_C**.

La ejecución del marco de activación de las siguientes tres funciones es muy similar a la mostrada en **matrizNxN_multiplicar_C**.

Finalmente para volver de la función **matrizNxN_operar_C** se pasa el resultado de terminos_no_cero por r0. De igual manera se aumenta el SP en 18 enteros para eliminar las variables locales de F[N][N] y E[N][N]. Para terminar se hace un **LDMIA** con r4-r11 y r14 que sustituye el PC por el LR.

Diferencias entre el marco de activación dado en AOC1 y en PH

Marco de activación en AOC1:

```
PUSH {lr, fp} // Decrementa en dos el valor de SP y luego transfiere el contenido del
operando fuente a la nueva dirección resultante en el registro recién modificado
MOV fp, sp // Movemos a FP el SP guardando de esta manera la posición inicial de la pila
PUSH {r2, r2, r0} // Pasamos los parámetros que usará la subrutina
sub sp, sp, #4 // Dejamos espacio para las variables como por ejemplo los resultados
```

En el marco de activación utilizado en AOC no teníamos en cuenta el estándar ATPCS por lo tanto no importaban qué registros se pusheaban ni en qué registros se pasaban los parámetros.

Marco de activación en PH:

Debido a que seguimos el estándar ATPCS debemos tener los parámetros de r0 a r3 y por lo tanto en esos registros se introducirá los valores que utilizaremos en las funciones.

Prólogo:

```
MOV IP, SP // Copia el valor actual del puntero de pila (SP) en el registro IP (R12). Esto
significa que después de ejecutar esta instrucción, el valor que estaba en SP ahora estará en IP
STMDB SP!, {r4-r10,FP,IP,LR,PC} // Store múltiple con decremento anterior del SP y
guardado de los registros
SUB FP, IP, #4
SUB SP, #SpaceForLocalVariables // Deja espacio para las variables locales
```


Análisis temporal

Para poder llevar a cabo el análisis temporal, se han recogido todas las métricas, en la siguiente tabla a continuación:

Función	Llamadas	Duración(μs)				Instrucciones			
		-O0	-O1	-O2	-O3	-O0	-O1	-O2	-O3
main	1	2.583	2.417	2.417	2.417	18	20	17	17
matrizNxN_verificar	1	10.417	9	8.25	8.25	69	26	20	20
matrizNxN_operar_C	1	33.083	23.471	22.667	23.083	62	56	55	56
matrizNxN_multiplicar_C	4	294.667	213.667	213.667	213.667	34	25	25	25
matrizNxN_sumar	1	19.250	13.750	13.750	13.750	25	19	19	19
matrizNxN_transponer	1	14.583	11.250	11.250	11.250	21	17	17	17
matriz3x3_operar_C_ARM	1	41.333	41.333	41.333	41.333	57	57	57	57
matriz3x3_operar_C_TOTAL	1	214.250	155.305	154.501	154.917				
matriz3x3_operar_C_ARM_TOTAL	1	188.667	148.167	148.167	148.167				
matriz3x3_operar_ARM	1	89.667	89.667	89.667	89.667	34	34	34	34
matriz3x3_operar_THUMB	1	116.167	116.167	116.167	116.167	63	63	63	63
matrizNxN_operar_C_equivalenteARM	1	90.5	75.917	75.917	75.917	47	40	47	40

Tabla sobre la diferencia entre las diferentes implementaciones de la función matriz_3x3_operar()

Cabe destacar que la sección inferior recogen la suma total de las funciones en C más sus elementos de ARM asociados.

Como podemos observar en las tablas respecto al tiempo, con el nivel de optimización -o0 observamos como las operaciones realizadas en C reducen su duración a medida que vamos aumentando el nivel de optimización (un claro ejemplo se encuentra en la función matrizNxN_operar_C) por otro lado, al comparar las diferentes versiones de la operación realizadas en THUMB, C_ARM y ARM comprobamos que la que se realiza en el menor tiempo posible es la ARM ya que su duración es únicamente de 89.667 μs siendo más rápida que su función en C original que tiene 154.917 μs de duración con nivel -o3. No obstante, nuestra función ARM no supera al equivalente real que hemos diseñado en C, teniendo este apenas 75.917 μs.

Del mismo modo se puede ver que la versión de THUMB es más lenta que la versión de ARM, ya que el objetivo de esta no es la velocidad sino la reducción de almacenamiento de instrucciones.

A modo de resumen, se podría decir que nuestra optimización de ARM, es más rápida que la versión original en C, incluso en los niveles -o2 y -o3. Sin embargo, si realizamos una versión equivalente al código ARM en C, nos supera a partir el nivel de optimización -o1, debido a que realiza optimizaciones relacionadas con desenrollar bucles.

Análisis espacial (comparación tamaño de instrucciones Thumb con ARM)

Por un lado, en el código de ARM optimizado hay un total de 34 instrucciones contando el epílogo y el prólogo. Teniendo en cuenta que cada instrucción ocupa 4 bytes da un total de 136 bytes.

Por otro lado, el código de THUMB cuenta con un total de 5 instrucciones ARM, y 58 instrucciones del repertorio THUMB. No obstante las instrucciones en THUMB ocupan solamente 2 bytes por lo que da en total el mismo tamaño de código que en ARM, 136 bytes.

Pese a que ocupen menos las instrucciones en THUMB, el rango de operaciones realizables es mucho más reducido por lo que acciones que se podrían realizar en 2 operaciones en THUMB tiene que ser en 5, por ejemplo

ARM

```
; A(r9) = i(r6) * 12 + k(r8) * 4 + @A(r0)
MLA    R9, R10, R6, R0
LDR    R9, [R9, R8, LSL #2]
```

THUMB

```
; A(r3) = i(r0) * 12 + k(r2) * 4 + @A(r8)
MOVS   R3, R0
MULS   R3, R7, R3          ; r3 = i(r0, r3)*12(r7)
ADD     R3, R3, R8          ; r3 = i(r3)*12(r7) + @A(r8)
LSLS   R7, R2, #2          ; r7 = 4 * k(r2)
LDR     R3, [R3, R7]
```

De la misma manera se podrían descontar los bytes que ocupan las instrucciones para entrada y salida del modo thumb ya que estos se podrían reducir sin utilizar el modo THUMB mezclado con ARM, y teniendo todo en THUMB.

Por lo tanto concluimos que en cuanto a rapidez nuestro diseño en ARM tiene mayor eficiencia pero en cuanto almacenamiento en memoria es mejor el THUMB.

A continuación se muestra un gráfico de las cuatro versiones de operar, en este podemos observar una relación total entre el número de bytes que ocupa el código y su coste temporal. Se puede ver claramente que el más reducido en tamaño es ARM junto a THUMB, no obstante el código realizado en C siendo equivalente al ARM, obtiene los mejores resultados temporales.

A pesar de esto, si juntamos el coste temporal junto con el tamaño del código, en proporción el código que hemos desarrollado en ARM, resulta con el mejor balance entre estos dos.

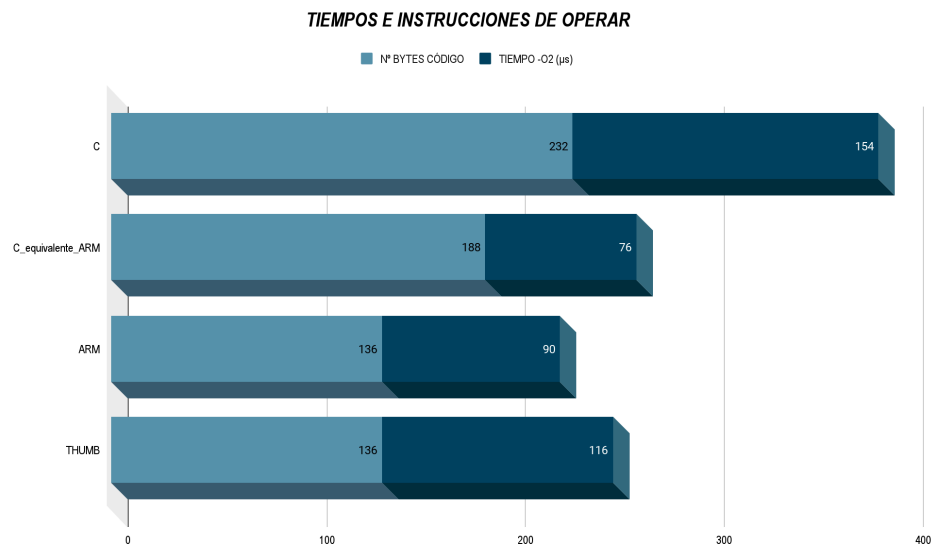


Gráfico sobre los tiempos de las distintas funciones implementadas y el tiempo en µs

CONCLUSIONES

Con la realización de esta práctica hemos podido observar como realizando la función operar de diferentes formas tiene ventajas y desventajas en cada una de ellas, al realizar `matriz3x3_operar_C` vemos como es un código portátil realizado en alto nivel que nos permite llevar el código a diferentes entornos o plataformas hardware. De esta manera podemos desarrollar programas legibles de forma más intuitiva y con mayor rapidez.

A continuación, en la `matriz3x3_operar_ARM` nuestro objetivo era la eficiencia temporal del programa, buscando ser más rápidos que el compilador, optimizando el código y finalmente venciendo a su equivalente realizado en C. Siendo el rendimiento de la función mucho mayor. Esto se ha conseguido no obstante solo ante la versión original de C. Al diseñar un programa en C equivalente aunque no tan intuitivo inicialmente, se ha podido observar que el compilador al empezar a optimizar supera a nuestra versión ARM. Se ha de añadir que muchas de las optimizaciones realizadas por el compilador, ya sea la técnica de desenrollar bucles, hacen que la legibilidad y escalabilidad del código se vea muy reducida.

En tercer lugar, al hacer `matriz3x3_operar_THUMB` hemos comprobado cómo se pueden reducir el tamaño en memoria utilizado, comparándolo con ARM y teniendo en cuenta que el número de instrucciones en THUMB es el doble que las utilizadas en ARM y aún así ocupan el mismo tamaño en memoria.

Finalmente, para concluir se podría decir que ninguna de las tres versiones en distintos lenguajes, es mejor o peor que las demás. Cada una destaca en unos aspectos y se peor en otros. Para escoger cuál es la mejor hay que fijarse en el contexto en el que te encuentres y ser capaz de valorar cuales pueden ser más beneficiosos.

ANEXOS

Anexo main.c

```
// fuentes Proyecto Hardware 2024
// version 0.1
//      Nekane Díaz Montoya      870795
//      Jorge Hernández Aznar 872838

#include "matriz_3x3.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Inicialización de matrices usando el tipo de datos Matriz3x3
static int Test_A[N][N] = {
    {1, 0, 0},
    {0, 1, 0},
    {0, 0, 1}
};

static int Test_B[N][N] = {
    {9, 8, 7},
    {6, 5, 4},
    {3, 2, 1}
};

/* *****
 * declaración funciones internas
 */

// funcion que ejecuta las distintas versiones de la implementacion en C, ARM y Thumb y verifica que dan el
// mismo resultado.
// recibe las matrices con las que operar
// devuelve si todas las versiones coinciden en el numero de terminos_no_cero o no y el resultado de la
// operacion.
uint8_t matrizNxN_verificar(int A[N][N], int B[N][N], int C[N][N], int D[N][N], int Resultado[N][N]);

// Dada una matriz A y devuelve en la matriz Transpuesta, el resultado de transponer A
void matrizNxN_transponer(int A[N][N], int Transpuesta[N][N]){
    int i,j;
    for(i = 0; i < N; i++){
        for(j = 0; j < N; j++){
            Transpuesta[j][i] = A[i][j];
        }
    }
}

// Dada una matriz A y otra B devuelve en Resultado la suma de estas
void matrizNxN_sumar(int A[N][N], int B[N][N], int Resultado[N][N]){
    int i,j;
    for(i = 0; i < N; i++){
        for(j = 0; j < N; j++){
            Resultado[i][j] = A[i][j] + B[i][j];
        }
    }
}

/* *****
 * IMPLEMENTACIONES
 */
// Dada una matriz A y otra B devuelve en Resultado la multiplicación de estas
void matrizNxN_multiplicar_C(int A[N][N], int B[N][N], int Resultado[N][N]){
    int i,j,k;
    for(i = 0; i < N; i++){
        for(j = 0; j < N; j++){
            for(k = 0; k < N; k++){
                Resultado[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

// función equivalente a matrizNxN_operar_C pero con un diseño monolítico equivalente al implementado en la
```

```

versión ARM
uint8_t matrizNxN_operar_C_equivalenteARM(int A[N][N], int B[N][N], int C[N][N], int D[N][N], int
Resultado[N][N]) {
    uint8_t terminos_no_cero = 9;
    int suma;
    int i,j,k;
    for(i = 0; i < N; i++){
        for(j = 0; j < N; j++){
            suma = 0;
            for(k = 0; k < N; k++){
                suma += A[i][k] * B[k][j] + C[j][k] * D[k][i];
            }
            Resultado[i][j] = suma;
            if (suma != 0) {
                terminos_no_cero--;
            }
        }
    }
    return terminos_no_cero;
}

//funcion que calcula Resultado = A*B + transpuesta(C*D) y devuelva el numero de terminos distintos de cero en
el Resultado
//ayudandose de funcion matrizNxN_multiplicar_C que calcula A*B de NxN
uint8_t matrizNxN_operar_C(int A[N][N], int B[N][N], int C[N][N], int D[N][N], int Resultado[N][N]){
    uint8_t terminos_no_cero;
    int E[N][N],F[N][N];
    int i,j;
    for(i = 0; i < N; i++){
        for(j = 0; j < N; j++){
            E[i][j] = 0;
            Resultado[i][j] = 0;
        }
    }

    matrizNxN_multiplicar_C(A, B, Resultado);
    matrizNxN_multiplicar_C(C, D, E);
    matrizNxN_transponer(E,F);
    matrizNxN_sumar(Resultado,F,Resultado);

    terminos_no_cero = 9;

    for(i = 0; i < N; i++){
        for(j = 0; j < N; j++){
            if(Resultado[i][j] == 0) {
                terminos_no_cero--;
            }
        }
    }
    return terminos_no_cero;
}

// Compara los resultados de operar una matriz 3x3 en distintas formas, C, Arm y Thumb y devuelve false(0) en
caso de que
// todas las funciones devuelvan el mismo número de terminos no cero , y true(1) en caso contrario
uint8_t matrizNxN_verificar(int A[N][N], int B[N][N], int C[N][N], int D[N][N], int Resultado[N][N]){
    uint8_t terminos_no_cero_C;
    uint8_t terminos_no_cero_ARM_C;
    uint8_t terminos_no_cero_ARM;
    uint8_t terminos_no_cero_THB;
    uint8_t terminos_no_cero_C_equivalenteARM;
    uint8_t resultado;

    terminos_no_cero_C = matrizNxN_operar_C(A, B, C, D, Resultado);
    terminos_no_cero_ARM_C = matriz3x3_operar_ARM_C(A, B, C, D, Resultado);
    terminos_no_cero_ARM = matriz3x3_operar_ARM(A, B, C, D, Resultado);
    terminos_no_cero_THB = matriz3x3_operar_THB(A, B, C, D, Resultado);
    terminos_no_cero_C_equivalenteARM = matrizNxN_operar_C_equivalenteARM(A, B, C, D, Resultado);

    resultado = !((terminos_no_cero_C != terminos_no_cero_ARM_C) || (terminos_no_cero_ARM_C !=
terminos_no_cero_ARM) || (terminos_no_cero_ARM != terminos_no_cero_THB) || (terminos_no_cero_THB !=
terminos_no_cero_C_equivalenteARM));

    return resultado;
}

// Compara todas las componentes de las matrices rC, rCArm, rArm, rThumb y rCOptimizado y devuelve fales(0)
si solo si todas la matrices son iguales,
// y devuelve true(1) en caso contrario
uint8_t matrizNxN_compararResultados(int rC[N][N], int rCArm[N][N], int rArm[N][N], int rThumb[N][N], int
rCOptimizado[N][N]) {

    uint8_t resultado = 0;
    int i, j;
    for(i = 0; i < N && resultado == 1; i++){
        for(j = 0; j < N && resultado == 1; j++){

```

```

        if( rC[i][j] != rCArm[i][j] || rCArm[i][j] != rArm[i][j] || rArm[i][j] !=
rThumb[i][j] || rThumb[i][j] != rCOptimizado[i][j]){
            resultado = 1;
        }
    }
    }
    return resultado;
}

// Devuelve un número aleatorio en el rango (a,b)
int randInt(int min, int max) {
    return min + rand() % (max - min +1);
}

// Asigna valores aleatorios a las matrices A, B, C y D
int leerMatrices(int A[N][N], int B[N][N], int C[N][N], int D[N][N]){
    int i, j;
    for(i = 0; i < N; i++){
        for(j = 0; j < N; j++) {
            A[i][j] = randInt(1,10);
            B[i][j] = randInt(1,10);
            C[i][j] = randInt(1,10);
            D[i][j] = randInt(1,10);
        }
    }
    return 0;
}

// Compara los resultados de operar una matriz 3x3 en distintas formas, C, Arm y Thumb y devuelve false(0) en
caso de que
// todas las funciones calculen el mismo resultado, y true(1) en caso contrario
uint8_t verificar_resultados(){
    int A[N][N], B[N][N], C[N][N], D[N][N];
    int rC[N][N], rCArm[N][N], rArm[N][N], rThumb[N][N], rCOptimizado[N][N];
    // Se inicializan las matrices A, B, C y D de forma aleatoria
    leerMatrices(A,B,C,D);
    // Se calculan sus resultados en distintas versiones
    matrizNxN_operar_C(A, B, C, D, rC);
    matriz3x3_operar_ARM_C(A, B, C, D, rCArm);
    matriz3x3_operar_ARM(A, B, C, D, rArm);
    matriz3x3_operar_THB(A, B, C, D, rThumb);
    matrizNxN_operar_C_equivalenteARM(A, B, C, D, rCOptimizado);
    // Se devuelve el resultado de comparar los resultados de distintas versiones
    return matrizNxN_compararResultados(rC, rCArm, rArm, rThumb, rCOptimizado);
}

// MAIN
int main (void) {
    int Resultado_E[N][N];
    int error;

    int Test_C[N][N] = {
        {1, 0, 2},
        {0, 1, 2},
        {2, 0, 1}
    };

    int Test_D[N][N] = {
        {2, 1, 0},
        {1, 2, 0},
        {0, 0, 2}
    };

    // Verificar resultados con terminos no cero original
    error = matrizNxN_verificar(Test_A, Test_B, Test_C, Test_D, Resultado_E);

    // Nueva función de verificación validando la igualdad de los resultados
    //error = verificar_resultados();

    while(1); //no hay S.O., no se retorna
}

```

Anexo matriz 3x3_operar_ARM_C.s

```

; Nekane Díaz Montoya 870795
; Jorge Hernández Aznar 872838
AREA codigo, CODE, READONLY
EXPORT matriz3x3_operar_ARM_C
IMPORT matrizNxN_multiplicar_C
PRESERVE8 {TRUE}

ENTRY
matriz3x3_operar_ARM_C
; PRÓLOGO
STMDB SP!, {R4-R12, LR} ; Se guardan los registros antiguos y el link

register para volver
MOV FP, SP
SUB SP, SP, #72 ; Se almacena espacio para las variables E y F, 36
bytes respectivamente

MOV R6, R2 ; r6 = @C
MOV R7, R3 ; r7 = @D
LDR R2, [FP, #40] ; R2 = @Resultado
ADD R4, SP, #40 ; R4 = @E

MOV R8, #0 ; i = r8
MOV R9, #0 ; R9 = 0 CTE

ini_for CMP R8, #9
BGE end_for

STR R9, [R2, R8, LSL #2]
STR R9, [R4, R8, LSL #2]

ADD R8, R8, #1
B ini_for

end_for

MOV R8, R0 ; R8 = @A
MOV R9, R1 ; R9 = @B
MOV R10, R2 ; R10 = @Resultado

; llamada a multiplicar(A,B,Resultado)
BL matrizNxN_multiplicar_C

MOV R0, R6 ; R0 = @C
MOV R1, R7 ; R1 = @D
MOV R2, R4 ; R2 = @E

; llamada a multiplicar(C,D,E)
BL matrizNxN_multiplicar_C

; Trasponer (E, F)
MOV R5, SP ; R5 = @F
MOV R11, #12

; R0 = i = 0
MOV R0, #0

trans_i_ini ; i < 3
CMP R0, #3
BGE trans_i_end
; R1 = j = 0
MOV R1, #0

trans_j_ini ; j < 3
CMP R1, #3
BGE trans_j_end

; @E = E + j*4 + i*12
MUL R12, R11, R0
ADD R2, R12, R1, LSL #2
LDR R3, [R4, R2]

; @F = F + i*4 + j*12
MUL R12, R11, R1
ADD R2, R12, R0, LSL #2
STR R3, [R5, R2]

ADD R1, R1, #1 ; i++
B trans_j_ini

trans_j_end
ADD R0, R0, #1 ; j++
B trans_i_ini

trans_i_end

; Sumar(Resultado,F,Resultado) + Check terminos_no_cero

```



```

; R10 = @Resultado      R5 = @F

MOV     R0, #9   ; R0 = terminos_no_cero
MOV     R1, #0   ; R1 = #0
MOV     R2, #0   ; R2 = i

ini_f_suma    CMP     R2, #9
               BGE     end_f_suma

               LDR     R3, [R5, R2, LSL #2]      ; R3 = F[i][j]
               LDR     R4, [R10, R2, LSL #2]     ; R4 = Resultado[i][j]
               ADD     R3, R4, R3
               CMP     R3, R1                    ; if (Resultado[i][j] == 0)
               SUBEQ   R0, R0, #1                ; terminos_no_cero--
               STR     R3, [R10, R2, LSL #2]

               ADD     R2, R2, #1
               B       ini_f_suma

end_f_suma

; EPÍLOGO
ADD     SP, SP, #72
LDMIA   SP!, {R4-R12, LR}
BX      LR

END

```

Anexo matriz 3x3_operar_ARM.s

```
; Nekane Diaz Montoya 870795
; Jorge Hernández Aznar 872838
AREA codigo, CODE, READONLY
EXPORT matriz3x3_operar_ARM
PRESERVE8 {TRUE}

ENTRY
matriz3x3_operar_ARM
; PRÓLOGO
STMDB SP!, {R4-R12, LR} ; Se guardan los registros antiguos y el link
register para volver
MOV FP, SP
LDR R4, [FP, #40] ; R4 = @Resultado

; multiplicar(A,B,Resultado) and multiplicar(C,D,E)
MOV R10, #12
MOV R14, #9
; for (i = 2; i >= 0; i--)
MOV R6, #2 ; R6 = i = 2
ini_f_i_m
; for (j = 2; j >= 0; j--)
MOV R7, #2 ; R7 = j = 2
ini_f_j_m
; En r5 se va a guardar el resultado de cada Resultado[i][j]
MOV R5, #0
; En r12 se va a guardar el valor de cada E[i][j]
MOV R12, #0
; for (k = 2; k >= 0; k--)
MOV R8, #2
ini_f_k_m
; A(r9) = i(r6) * 12 + k(r8) * 4 + @A(r0)
MLA R9, R10, R6, R0
LDR R9, [R9, R8, LSL #2]
; B(r11) = k(r8) * 12 + j(r7) * 4 + @B [r1]
MLA R11, R10, R8, R1
LDR R11, [R11, R7, LSL #2]
; Resultado[i][j] += A[i][k] * B[k][j] + Resultado[i][j]
MLA R5, R9, R11, R5

; C y D se calculan directamente transpuestas para que al multiplicarlas se obtenga
E transpuesta
; Para ello en vez de hacer E[i][j] += C[j][k] * D[k][i] + E[i][j] en vez de
E[i][j] += C[i][k] * D[k][j] + E[i][j]
; C(r9) = j(r7) * 12 + k(r8) * 4 + @C(r2)
MLA R9, R10, R7, R2
LDR R9, [R9, R8, LSL #2]
; D(r11) = k(r8) * 12 + i(r6) * 4 + @D(r3)
MLA R11, R10, R8, R3
LDR R11, [R11, R6, LSL #2]
; E[i][j] += C[j][k] * D[k][i] + E[i][j]
MLA R12, R9, R11, R12

SUBS R8, R8, #1 ; R8--
BPL ini_f_k_m
; fin_for_k

ADD R5, R5, R12 ; RESULTADO[i][j] + E[i][j]
CMP R5, #0 ; if (Resultado[i][j] == 0)
SUBEQ R14, R14, #1 ; terminos_no_cero--

MLA R11, R10, R6, R4
STR R5, [R11, R7, LSL #2]

SUBS R7, R7, #1 ; R7--
BPL ini_f_j_m
; fin_for_j
SUBS R6, R6, #1 ; R6--
BPL ini_f_i_m
; fin_for_i

; end multiplicar(A,B,Resultado) and multiplicar(C,D,E)

MOV R0, R14

; EPÍLOGO
LDMIA SP!, {R4-R12, LR}
BX LR

END
```

Anexo matriz3x3_operar_THB

```

; Nekane Díaz Montoya 870795
; Jorge Hernández Aznar 872838
AREA codigo, CODE, READONLY
EXPORT matriz3x3_operar_THB
PRESERVE8 {TRUE}
ENTRY
matriz3x3_operar_THB
; PRÓLOGO
ARM
STMDB SP!, {R4-R12, LR} ; Se guardan los registros antiguos y el link
register para volver

ADR R4, ini_thumb + 1
BX R4

THUMB
ini_thumb
LDR R4, [SP, #40] ; R12 = @Resultado, #40
MOV R12, R4

; No hace falta inicializar Resultado ni E si solo vamos a hacer stores en memoria
; Movemos las direcciones de las matrices a high registers
MOV R8, R0 ; movemos @A a r8
MOV R9, R1 ; movemos @B a r9
MOV R10, R2 ; movemos @C a r10
MOV R11, R3 ; movemos @D a r11

; multiplicar(A,B,Resultado) and multiplicar(C,D,E)
MOVS R0, #0
MOV R14, R0 ; R14 = terminos_no_cero

; for (i = 2; i >= 0; i--)
MOVS R0, #2 ; R0 = i = 2
ini_f_i_m

; for (j = 2; j >= 0; j--)
MOVS R1, #2 ; R1 = j = 2
ini_f_j_m

; En r5 se va a guardar el resultado de cada Resultado[i][j]
MOVS R5, #0
; En r6 se va a guardar el valor de cada E[i][j]
MOVS R6, #0

; for (k = 2; k >= 0; k--)
MOVS R2, #2 ; R2 = k = 2
ini_f_k_m

MOVS R7, #12
; A(r3) = i(r0) * 12 + k(r2) * 4 + @A(r8)
MOVS R3, R0
MULS R3, R7, R3 ; r3 = i(r0, r3)*12(r7)
; B(r4) = k(r8) * 12 + j(r7) * 4 + @B(r9)
MOVS R4, R2
MULS R4, R7, R4 ; r4 = k(r2, r4)*12(r7)

ADD R3, R3, R8 ; r3 = i(r3)*12(r7) + @A(r8)
LSLS R7, R2, #2 ; r7 = 4 * k(r2)
LDR R3, [R3, R7] ; guarda lo que hay en la direccion
de A apuntada en r3

ADD R4, R4, R9 ; r4 = k(r2, r4)*12(r7)
LSLS R7, R1, #2 ; r7 = 4 * j(r1)
LDR R4, [R4, R7] ; guarda lo que hay en la direccion de B
apuntada en r4

; Resultado[i][j] += A[i][k] * B[k][j] + Resultado[i][j]
MULS R4, R3, R4 ; r4 = A[i][j](r3) * B[i][j](r4)
ADDS R5, R5, R4 ; r5 = r4 + Resultado[i][j]

MOVS R7, #12
; C(r3) = j(r1) * 12 + k(r2) * 4 + @C(r10)
MOVS R3, R1
MULS R3, R7, R3 ; r3 = i(r3, r0) * 12(r7)

; D(r11) = k(r2) * 12 + i(r0) * 4 + @D(r11)
MOVS R4, R2
MULS R4, R7, R4 ; r4 = k(r4, r2) * 12(r7)

ADD R3, R3, R10 ; i(r3, r0) * 12(r7) + @C(r10)
LSLS R7, R2, #2 ; r7 = 4 * k(r2)
LDR R3, [R3, R7] ; guarda lo que hay en la direccion
de C apuntada en r3

ADD R4, R4, R11 ; k(r4, r2) * 12(r7) + @D

```

```

                                LSLS    R7, R0, #2                ; r7 = 4 * j(r1)
                                LDR     R4, [R4, R7]            ; guarda lo que hay en la direccion de D
apuntada en r4

                                ; E[i][j] += C[i][k] * D[k][j]    + E[i][j]
                                MULS    R4, R3, R4            ; r4 = C[i][j](r3) * D[i][j](r4)
                                ADDS    R6, R6, R4            ; r6 = r4 + E[i][j]

                                SUBS    R2, R2, #1            ; R2--
                                BPL     ini_f_k_m
                                ; fin_for_k
                                MOVS    R2, #1                ; R2 = #1
                                MOVS    R7, #12

                                ADDS    R5, R5, R6            ; Resultado[i][j] + E[i][j]
                                CMP     R5, #0                ; if (Resultado[i][j] != 0){
                                BEQ     es_cero
                                ADD     R14, R14, R2          ; terminos_no_cero++}

es_cero                        MULS    R7, R0, R7            ; R7 = i * 12 (porque cada fila tiene 12
bytes, 3 enteros de 4 bytes)
                                ADD     R7, R7, R12           ; R7 = i * 12 + @Resultado (R12)
                                LSLS    R4, R1, #2            ; R4 = j * 4 (columna multiplicada por 4 bytes
por elemento)
                                                                ; R7 = i * 12 +
                                @Resultado + j * 4
                                STR     R5, [R7, R4]           ; Almacena el valor en la dirección calculada

                                SUBS    R1, R1, #1            ; R1--
                                BPL     ini_f_j_m
                                ; fin_for_j
                                SUBS    R0, R0, #1            ; R0--
                                BPL     ini_f_i_m
                                ; fin_for_i

                                MOV     R0, R14                ; R0 = terminos_no_cero

                                ADR     R1, ini_arm
                                BX      R1
                                ARM

ini_arm
                                ; EPÍLOGO
                                LDMIA   SP!, {R4-R12, LR}
                                BX      LR

                                END

```