



# Universidad Nacional Autónoma de México



## Facultad de Ingeniería

### Integrantes:

Espinoza Matamoros Percival Ulises - 320025561

Flores Colin Victor Jaziel - 320266083

Lara Hernandez Angel Husiel - 320060829

## Laboratorio de Microcomputadoras

Grupo: 06 - Semestre: 2026-2

### Practica 2:

Programación en Ensamblador. Direccionamiento  
Indirecto

### Profesor:

Ing. Moises Melendez Reyes

### Fecha de Entrega:

8 de Marzo del 2026



## 1. Objetivo:

Programar las variantes del modo de direccionamiento indirecto existentes para los procesadores ARM.

### Actividad 1

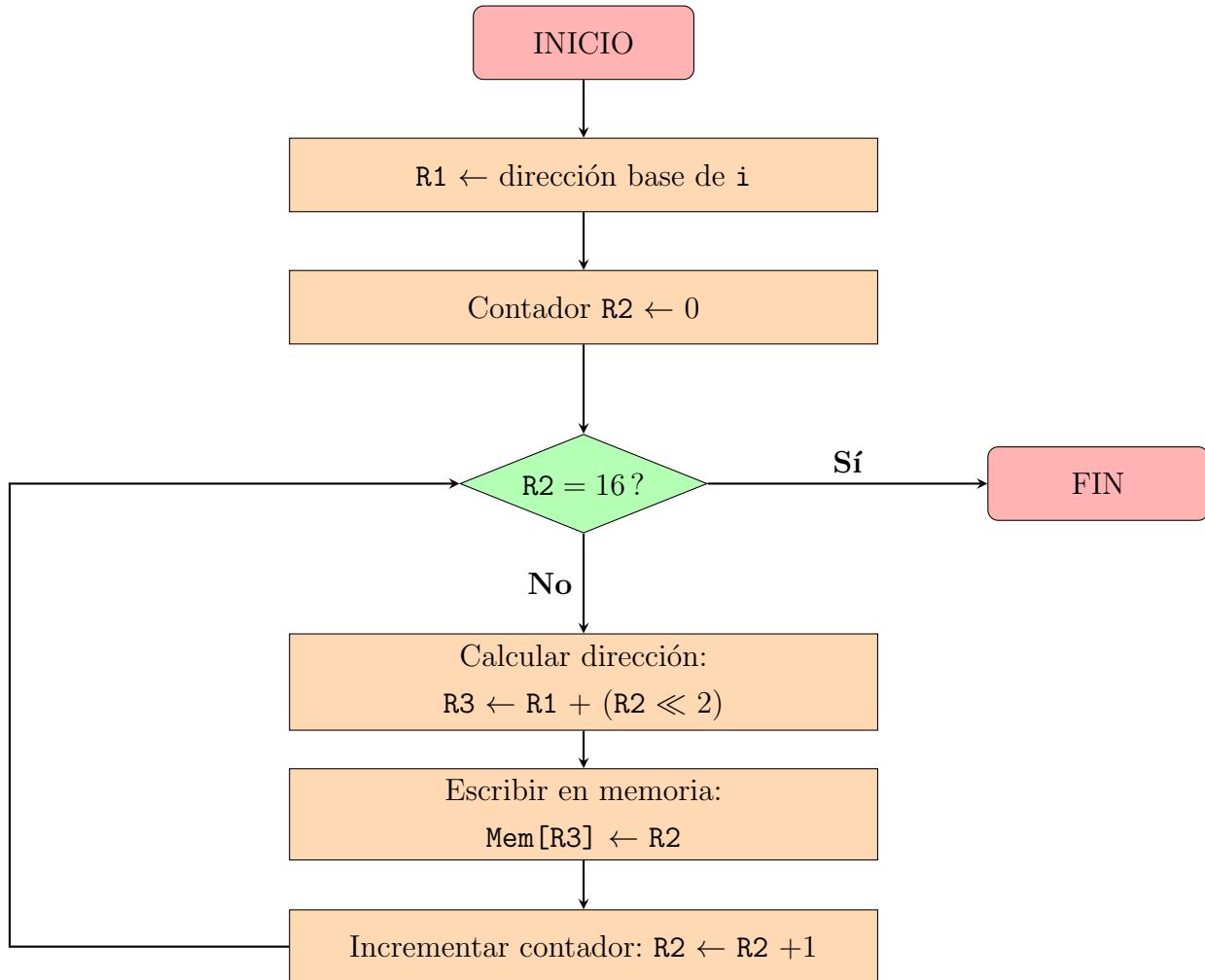
Escribir, comentar, compilar y comprobar el funcionamiento del siguiente programa, cuyo objetivo es guardar el valor de un contador en un arreglo de 16 posiciones de 32 bits utilizando desplazamientos lógicos.

### Propuesta de solución

El programa implementa una estructura de bucle para llenar un bloque de memoria reservado de forma dinámica de 64 bytes lo que equivale a 16 palabras de 4 bytes cada una y el eje principal del cálculo se basa en el desplazamiento de registros para corrimientos de bits.

Para acceder a cada posición del arreglo sin modificar el apuntador base (**R1**), se emplea la instrucción **ADD** con un desplazamiento lógico a la izquierda (**LSL #2**). Dado que cada elemento ocupa 4 bytes ( $2^2$ ), desplazar el índice contenido en el contador **R2** dos bits a la izquierda equivale matemáticamente a multiplicarlo por 4 y este valor resultante se suma a la dirección base para obtener la dirección exacta de destino en **R3**, donde se almacena el valor del contador mediante la instrucción **STR**.

A continuación se presenta el diagrama de flujo correspondiente al algoritmo descrito:



## Desarrollo

Listing 1: Código de la Actividad 1

```

1  /* ACTIVIDAD 1: Direcccionamiento con desplazamiento a la izquierda (LSL)
2      Objetivo: Guardar el valor del contador en un arreglo de 16
3          posiciones.
4 */
5  .data
6      i: .skip 64           @ Reserva 64 bytes de memoria (16
7          palabras de 4 bytes)
  
```



```
8 .global main           @ Define 'main' como global para Code
  ::Blocks / Linker

9

10 main:
11   ldr r1, =i           @ Carga en R1 la dirección base de la
                         variable 'i'
12   mov r2, #0           @ R2 será nuestro contador,
                         inicializado en 0

13
14 loop:
15   cmp r2, #16          @ Compara el contador R2 con el límite
                         de 16
16   beq fin              @ Si R2 es igual a 16 (Branch if EQUAL
                         ), salta a la etiqueta 'fin'

17
18   add r3, r1, r2, LSL #2    @ R3 = R1 + (R2 desplazado a la
                             izquierda 2 bits). Equivale a R3 = R1 + (R2 * 4). Calcula la
                             dirección en memoria.
19   str r2, [r3]          @ Guarda el valor actual del contador
                         (R2) en la dirección de memoria apuntada por R3
20   add r2, r2, #1          @ Incrementa el contador (R2 = R2 + 1)
21   b loop                @ Salto incondicional (Branch) de
                         regreso a 'loop'

22
23 fin:
24   MOV R7, #1            @ Carga la llamada al sistema sys_exit
                         (1)
25   SVC 0                 @ Ejecuta la llamada para salir
                         limpiamente al SO
```



The screenshot shows the Code::Blocks IDE interface. The left pane displays the assembly code for the file `ps2.s`. The code defines a global variable `i`, initializes it to 0, and then enters a loop where it increments `i` by 1, checks if it equals 10, and if not, loops back. The right pane shows the CPU Registers window with the following values:

Register	Hex	Interpreted
r0	0x0	0x1
r1	0x41813274	1090902112
r2	0x4181327C	1090902140
r3	0x40000488	3273743880
r4	0x418132A8	1090901800
r5	0x0	0x0
r6	0x40000209	3273742809
r7	0x0	0x0
r8	0x0	0x0
r9	0x0	0x0
r10	0x40011000	3273811456
r11	0x0	0x0
r12	0x418132210	1090902104
sp	0x41804000	3273742100
lr	0x41804001	3273741457
pc	0x40000040	<main>
cpar	0x40000010	3274059344
tpsr	0x0	0x0
tpslid	0x41804000	3273743712
tpsrc	0x40000000	3273741624
AFSR0_E1L1	0x0	0x0
AFSR1_E1L1	0x0	0x0
000010R	0x3515F021	890630177
000024R	0x0	0x0

The assembly code and registers are identical to Figure 1.

Figura 1: Punto de interrupción inicial. Se observa la carga de la dirección base del arreglo en R1.

The screenshot shows the Code::Blocks IDE interface. The left pane displays the assembly code for the file `ps2.s`. The code defines a global variable `i`, initializes it to 0, and then enters a loop where it increments `i` by 1, checks if it equals 10, and if not, loops back. The right pane shows the CPU Registers window with the following values:

Register	Hex	Interpreted
r0	0x0	0x1
r1	0x418132A0	3273811200
r2	0x4181327C	1090902140
r3	0x40000488	3273743880
r4	0x418132A8	1090901800
r5	0x0	0x0
r6	0x40000209	3273742809
r7	0x0	0x0
r8	0x0	0x0
r9	0x0	0x0
r10	0x40011000	3273811456
r11	0x0	0x0
r12	0x418132210	1090902104
sp	0x41804000	3273742100
lr	0x41804001	3273741457
pc	0x40000040	<main>
cpar	0x40000010	3274059344
tpsr	0x0	0x0
tpslid	0x41804000	3273743712
tpsrc	0x40000000	3273741624
AFSR0_E1L1	0x0	0x0
AFSR1_E1L1	0x0	0x0
000010R	0x3515F021	890630177
000024R	0x0	0x0

The assembly code and registers are identical to Figure 1.

Figura 2: Inicio del bucle. El contador R2 inicia en 0 y se evalúa la condición de salto.



The screenshot shows the Code::Blocks IDE interface. The main window displays an assembly file named ps2.s. The code implements a loop that increments a value at memory location R1 by 4 in each iteration, starting from R2=0. The loop exits when R2 reaches 16. The CPU Registers panel shows the state of the processor registers after execution. The PC register points to the instruction following the loop's exit point.

```
.data
    .skip 64          @ Reserva 64 bytes de memoria (16 palabras de 4 bytes)
    i: .skip 16
.text
    .global main
main:
    ldr r1, =i        @ Carga en R1 la dirección base de la variable 'i'
    mov r2, #0         @ R2 será nuestro contador, inicializado en 0
loop:
    cmp r2, #16       @ Compara el contador R2 con el límite de 16
    beq fin           @ Si R2 es igual a 16 (Branch if Equal), salta a la etiqueta 'fin'
    add r3, r1, r2, LSL #2   @ R3 = R1 + (R2 desplazado a la izquierda 2 bits). Equivale a R3 = R1 + (R2 * 4). Calcula la dirección de memoria apuntada por R3
    str r2, [r3]       @ Guarda el valor actual del contador (R2) en la dirección de memoria apuntada por R3
    add r2, r2, #1      @ Incrementa el contador (R2 = R2 + 1)
    b loop            @ Salto incondicional (Branch) de regreso a 'loop'
fin:
    MOV R7, #1         @ Carga la llamada al sistema sys_exit (1)
    SVC 0             @ Ejecuta la llamada para salir limpiamente al SO
```

Register	Hex	Interpreted
R0	0x00000000	00000000
R1	0x400011040	1073811520
R2	0x00000000	00000000
R3	0x400011040	1073811520
R4	0x4181222F4	1090931500
R5	0x00000000	00000000
R6	0x400000000	1073742800
R7	0x00000000	00000000
R8	0x00000000	00000000
R9	0x00000000	00000000
R10	0x400011000	1073811456
R11	0x00000000	00000000
R12	0x4181222F4	1090931500
sp	0x4181222F0	1090931500
lr	0x4181222F4	1090931500
pc	0x400000000	1073742800 <loop>
cpar	0x000000000	2146566128
fpscr	0x00000000	00000000
fpcr	0x4181222F0	1090931500
fpcrld	0x400000000	1073741824
AFSR0_E1	0x00000000	00000000
AFSR1_E1	0x00000000	00000000
DBCSR	0x3515F01	899630177
DBCSRAR	0x00000000	00000000

Figura 3: Proceso de incremento. El registro PC apunta al retorno del ciclo tras aumentar R2.

This screenshot shows the assembly code ps2.s running in the Code::Blocks IDE. The code increments R2 until it reaches 16, at which point it exits the loop and calls sys\_exit(1). The CPU Registers panel shows the final state of the registers after the program has completed its execution.

```
.data
    .skip 64          @ Reserva 64 bytes de memoria (16 palabras de 4 bytes)
    i: .skip 16
.text
    .global main
main:
    ldr r1, =i        @ Carga en R1 la dirección base de la variable 'i'
    mov r2, #0         @ R2 será nuestro contador, inicializado en 0
loop:
    cmp r2, #16       @ Compara el contador R2 con el límite de 16
    beq fin           @ Si R2 es igual a 16 (Branch if Equal), salta a la etiqueta 'fin'
    add r3, r1, r2, LSL #2   @ R3 = R1 + (R2 desplazado a la izquierda 2 bits). Equivale a R3 = R1 + (R2 * 4). Calcula la dirección de memoria apuntada por R3
    str r2, [r3]       @ Guarda el valor actual del contador (R2) en la dirección de memoria apuntada por R3
    add r2, r2, #1      @ Incrementa el contador (R2 = R2 + 1)
    b loop            @ Salto incondicional (Branch) de regreso a 'loop'
fin:
    MOV R7, #1         @ Carga la llamada al sistema sys_exit (1)
    SVC 0             @ Ejecuta la llamada para salir limpiamente al SO
```

Register	Hex	Interpreted
R0	0x00000000	00000000
R1	0x400011040	1073811520
R2	0x00000010	00000010
R3	0x400011040	1073811520
R4	0x4181222F4	1090931500
R5	0x00000000	00000000
R6	0x400000000	1073742800
R7	0x00000000	00000000
R8	0x00000000	00000000
R9	0x00000000	00000000
R10	0x400011000	1073811456
R11	0x00000000	00000000
R12	0x4181222F4	1090931500
sp	0x4181222F0	1090931500
lr	0x4181222F4	1090931500
pc	0x400000000	1073742800 <fin>
cpar	0x000000000	2146566128
fpcr	0x4181222F0	1090931500
fpcrld	0x400000000	1073741824
AFSR0_E1	0x00000000	00000000
AFSR1_E1	0x00000000	00000000
DBCSR	0x3515F01	899630177
DBCSRAR	0x00000000	00000000

Figura 4: Finalización de la ejecución. El contador que es R2 alcanza el valor 0x10 (16) y el programa sale del ciclo.

## Análisis de resultados

Al iniciar la ejecución, como se muestra en la **Figura 1**, el registro R1 recibe la dirección base del arreglo *i*, la cual corresponde a 0x418122F4 en la memoria del sistema y el contador R2 se inicializa en 0 para controlar las 16 iteraciones propuestas.



---

Una vez dentro de la etiqueta `loop`, el programa realiza la primera iteración y al ser `R2 = 0` y utiliza un direccionamiento calculado mediante **Barrel Shifter** donde ocurre el primer desplazamiento `LSL #2` que no altera el valor base, por lo que `R3` apunta directamente al inicio del arreglo. Sin embargo, en iteraciones posteriores (como se sugiere en la **Figura 3**), cuando `R2` incrementa, el desplazamiento permite saltar de 4 en 4 bytes. Por ejemplo, cuando `R2 = 1`, la operación calcula `R1 + (1 × 4)`, posicionando `R3` en la siguiente palabra de memoria.

Finalmente, como se ve en la **Figura 4**, el ciclo termina correctamente cuando `R2` alcanza el valor hexadecimal `0x10`. La bandera de cero (`Z`) se activa tras la comparación `CMP` lo que permite que la instrucción `BEQ` transfiera el control a la etiqueta `fin` y el programa concluye invocando la llamada al sistema `SVC 0` con el código de salida `1` en `R7`, garantizando que los 16 valores (del `0` al `15`) han sido almacenados secuencialmente en la memoria reservada.



## Actividad 2

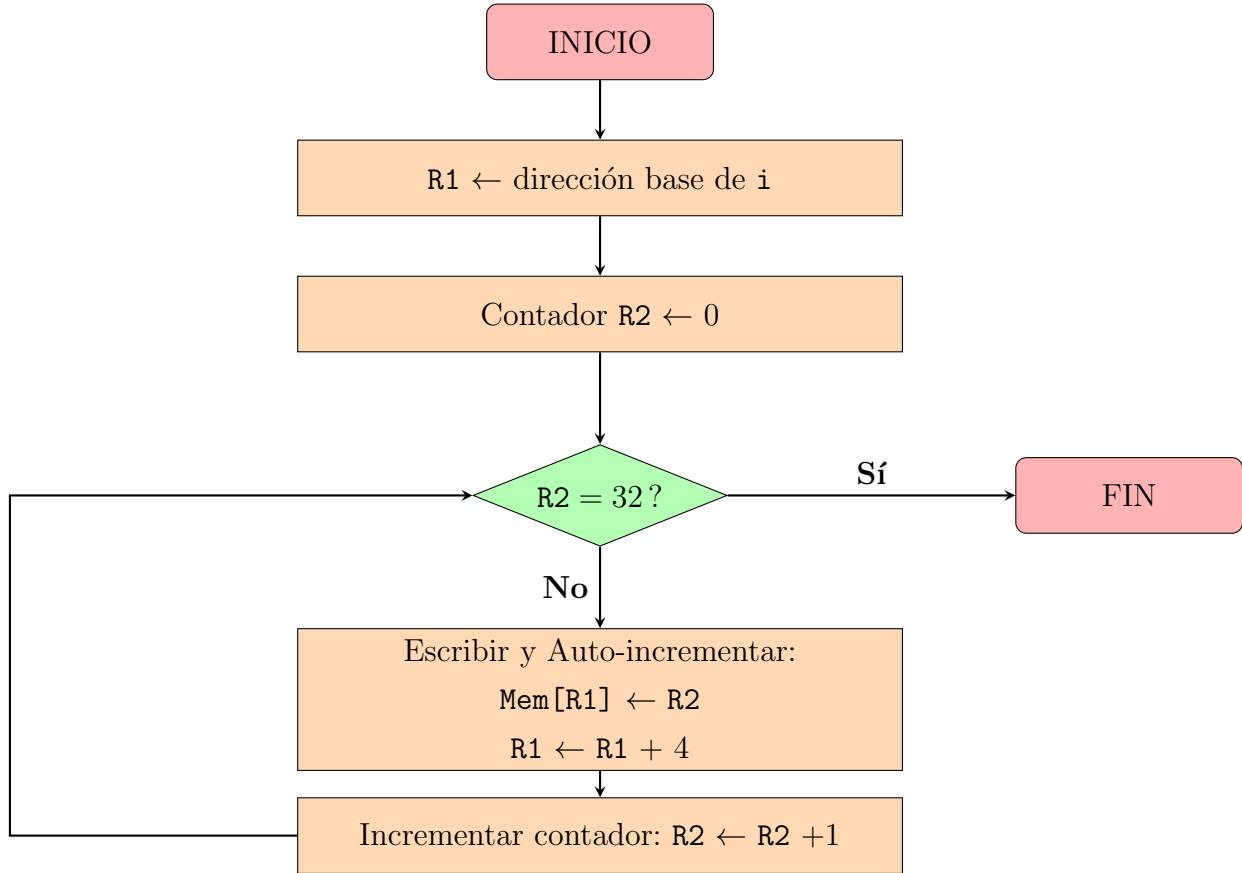
Modificar el programa de la actividad 1 para utilizar un modo de direccionamiento indexado distinto, aumentando la capacidad de almacenamiento al doble de datos (32 elementos).

### Propuesta de solución

Para esta actividad se tomó como base el código de la actividad 1 y se aplicó un **direcciónamiento post-indexado** y a diferencia de la Actividad 1, donde se calculaba la dirección en un registro auxiliar mediante desplazamientos, el direccionamiento post-indexado permite realizar la escritura en memoria y la actualización del apuntador en una sola instrucción.

En esta ocasión se reservan 128 bytes de memoria en vex de 64 en la sección **.data** para albergar 32 elementos de 4 bytes cada uno. El algoritmo utiliza el registro R1 como apuntador base y R2 como contador. En cada iteración, la instrucción **STR R2, [R1], #4** realiza dos acciones: primero, almacena el valor del contador en la dirección actual de R1; segundo, suma inmediatamente un valor inmediato de 4 a R1 para que apunte a la siguiente palabra lo que reduce el número de instrucciones aritméticas necesarias dentro del ciclo y simplifica la lógica del programa.

A continuación se presenta el diagrama de flujo que describe el funcionamiento del programa con direcciónamiento post-indexado:



## Desarrollo

Listing 2: Código de la Actividad 2

```

1  /* ACTIVIDAD 2: Direccionamiento Post-indexado con 32 datos
2   Objetivo: Guardar 32 números usando auto-incremento de dirección.
3 */
4 .data
5   i: .skip 128           @ Reserva 128 bytes (32 elementos * 4
6   bytes cada uno)
7
8 .text
9 .global main
10
11 main:
12   ldr r1, =i            @ Carga en R1 la dirección base del
13   arreglo 'i'
  
```



```
12      mov r2, #0          @ R2 es el contador, inicia en 0
13
14 loop2:
15     cmp r2, #32         @ Compara el contador con 32 (el doble
16                 que la act. 1)
17     beq salir           @ Si llegamos a 32, salta a 'salir'
18
19     str r2, [r1], #4      @ DIRECCIONAMIENTO POST-INDEXADO:
20                 Guarda R2 en la memoria de R1, y LUEGO suma 4 a R1 automá
21                ticamente.
22     add r2, r2, #1        @ Incrementa el contador R2 en 1
23     b loop2              @ Repite el bucle
24
25
26 salir:
27     MOV R7, #1          @ Prepara sys_exit
28     SVC 0                @ Termina ejecución
```

The screenshot shows the Code::Blocks IDE interface during assembly-level debugging. The main window displays the assembly code for the file ps2.s. The code includes instructions for initializing R2, entering a loop to increment R2 and update memory, and exiting via sys\_exit. The Registers window on the right shows the current state of the CPU registers, with R1 containing the value 0x40011040. The assembly code window has several red circles highlighting specific memory locations, notably R1 and the memory address 0x40011040 where the value 1 is stored.

Figura 5: Estado inicial de la depuración. R1 contiene la dirección del arreglo i 0x40011040



The screenshot shows the Code-Blocks IDE interface. The left pane displays the assembly code for the file `ps2.s`. The code initializes R1 to 1, R2 to 0, and then enters a loop where it increments R2 by 4 and checks if R2 reaches 32. If R2 reaches 32, it exits the loop. The right pane shows the CPU Registers window, which lists the values of various registers: R0 (0x0), R1 (0x40011040), R2 (0x0), R3 (0x40000408), R4 (0x41022240), R5 (0x0), R6 (0x40000309), R7 (0x0), R8 (0x0), R9 (0x0), R10 (0x40011000), R11 (0x0), R12 (0x41022240), R13 (0x40000309), R14 (0x41022240), R15 (0x40000309), R16 (0x41022240), R17 (0x40000309), R18 (0x41022240), R19 (0x40000309), R20 (0x41022240), R21 (0x40000309), R22 (0x41022240). The assembly code is as follows:

```
1 .data
  1: .skip 128      @ Reserva 128 bytes (32 elementos * 4 bytes cada uno)
  2 .
  3 .text
  4 .global main
  5
  6 main:
  7   ldr r1, =i      @ Carga en R1 la dirección base del arreglo 'i'
  8   mov r2, #0      @ R2 es el contador, inicia en 0
  9   loop:
 10    cmp r2, #32    @ Compara el contador con 32 (el doble que la act. 1)
 11    beq salir      @ Si llegamos a 32, salta a 'salir'
 12
 13    str r2, [r1], #4 @ DIRECCIONAMIENTO POST-INDEXADO: Guarda R2 en la memoria de R1, y LUEGO suma 4 a R1 automatica
 14    add r2, r2, #1   @ Incrementa el contador R2 en 1
 15    b loop2        @ Repite el bucle
 16
 17 salir:
 18   mov r7, #1      @ Prepara sys_exit
 19   svc 0           @ Termina ejecución
 20
 21
 22
```

Figura 6: El contador del programa que es el registro R2 inicia en 0.

This screenshot shows the same setup as Figure 6, but during the fifth iteration of the loop. The CPU Registers window now shows R1 at 0x40011054 and R2 at 0x5. The assembly code remains identical to Figure 6.

Figura 7: Quinta iteración del ciclo. El contador R2 tiene el valor 0x5 y el puntero R1 ha avanzado proporcionalmente a 0x40011054.

Figura 8: Fin de la ejecución. El contador R2 alcanza 0x20 (32 decimal), rompiendo la condición y finalizando el programa.

## Análisis de resultados

El análisis dinámico mediante el depurador confirma la eficiencia del direccionamiento post-indexado. Como se observa en la **Figura 1** el programa al inicio tiene el registro R1 apuntando a la dirección de memoria 0x40011040 que corresponde al arreglo i. A diferencia de la actividad anterior, donde se requería un registro adicional (R3) para calcular la dirección de cada elemento, aquí se trabaja directamente sobre el puntero base.

Al ejecutar la instrucción `str r2, [r1], #4`, el procesador realiza la escritura del valor contenido en R2 y, acto seguido actualiza R1. Este comportamiento se verifica en la **Figura 3**, donde tras varias iteraciones el valor de R1 ha incrementado constantemente en saltos de 4 bytes. Por ejemplo, cuando el contador R2 llega a 5, el puntero R1 ya se ha desplazado 20 bytes desde su posición original (0x40011054), demostrando el auto-incremento exitoso.

Finalmente, como se aprecia en la **Figura 4**, el bucle se rompe en el momento en que R2 iguala a 32 o (0x20 en hexadecimal). En este punto, la instrucción **BEQ** detecta que la comparación es verdadera y salta a la etiqueta **salir**. El resultado en memoria es un arreglo de 32 elementos que contiene la secuencia numérica del 0 al 31, logrando el doble de datos que la práctica anterior con un código más compacto y optimizado.



## Actividad 3

Realizar un programa almacene en memoria un arreglo de datos de 32 bits con 16 elementos; una vez transferidos, realizar la copia en sentido inverso en otro arreglo.

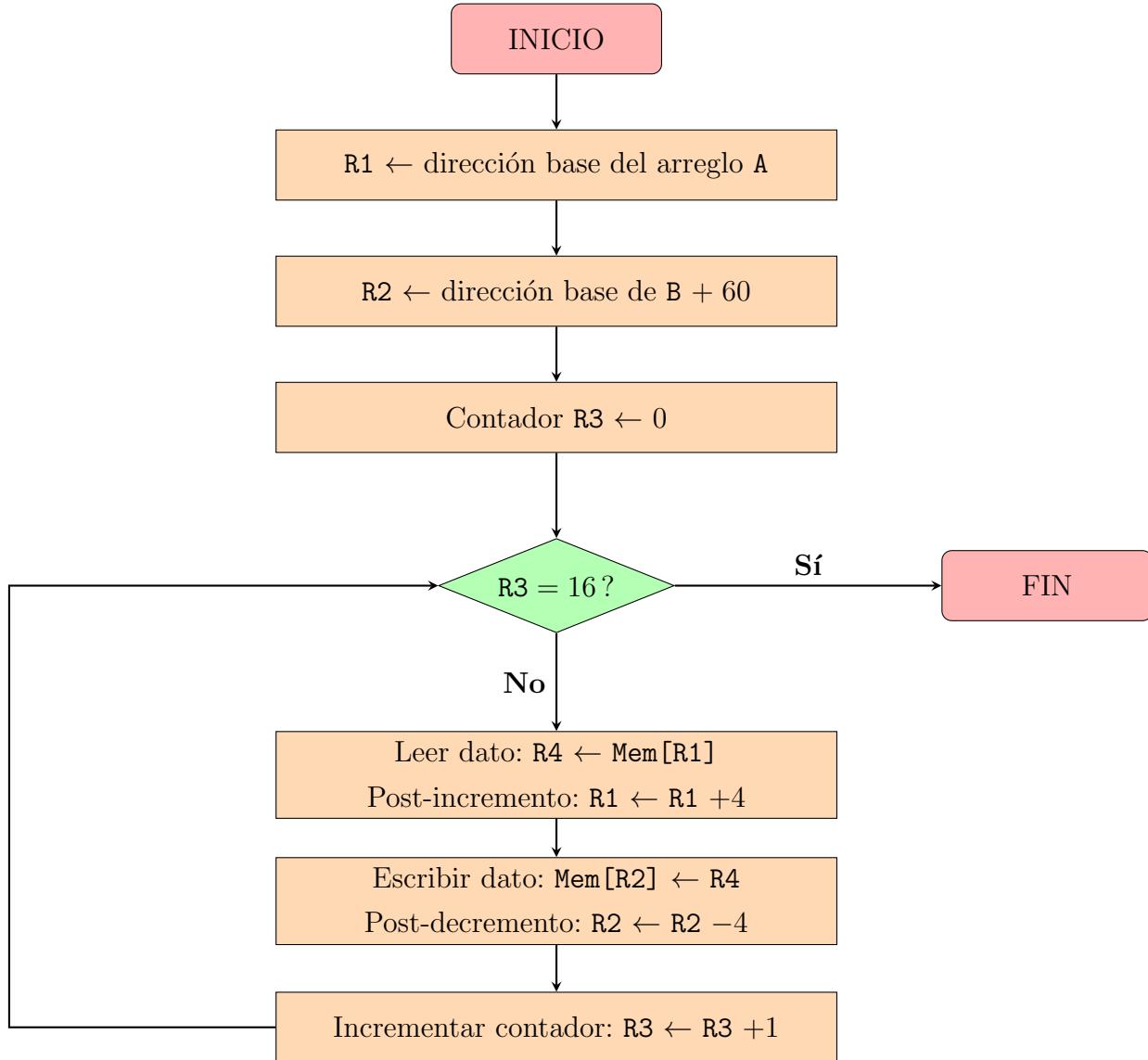
$$A = [\text{dato}_1, \text{dato}_2, \text{dato}_3, \text{dato}_4, \dots, \text{dato}_{15}, \text{dato}_{16}] \quad @\text{Original}$$

$$B = [\text{dato}_{16}, \text{dato}_{15}, \text{dato}_{14}, \text{dato}_{13}, \dots, \text{dato}_2, \text{dato}_1] \quad @\text{Copia}$$

### Propuesta de solución

Para resolver el problema se emplea una estrategia basada en dos apuntadores que recorren los arreglos en sentidos opuestos, aprovechando el modo de direccionamiento **post-indexado** de la arquitectura ARM. El apuntador de lectura R1 se inicializa en el primer elemento del arreglo A y avanza de forma **ascendente** (+4 bytes por iteración), mientras que el apuntador de escritura R2 se posiciona en la *última celda reservada* del arreglo destino B desplazándose 60 bytes desde su base, y retrocede de forma **descendente** (-4 bytes por iteración). Un contador R3, inicializado en cero, controla el número de iteraciones del ciclo; cuando su valor alcanza 16, la instrucción CMP activa la bandera Z del registro de estado y la instrucción BEQ transfiere el control fuera del bucle, finalizando la ejecución mediante una llamada al sistema (SVC 0). De esta forma, cada elemento leído secuencialmente de A se escribe en la posición inversa correspondiente de B, logrando la copia invertida sin consumir registros adicionales para el cálculo de direcciones.

A continuación se presenta el diagrama de flujo correspondiente al algoritmo descrito:



Se cargan las direcciones base de los arreglos mediante LDR: R1 queda apuntando al primer elemento de A; R2 recibe la dirección base de B y se desplaza +60 bytes para posicionarse en la última celda (índice 15), y el contador R3 se pone a cero. A continuación inicia el **ciclo principal**: al comienzo de cada iteración se evalúa la condición de salida  $R3 = 16$ ; si es **falsa**, la instrucción LDR con post-incremento lee el siguiente dato de A en R4 y adelanta R1 en +4 bytes, luego la instrucción STR con post-decremento escribe R4 en la posición actual de B y retrocede R2 en -4 bytes, efectuando el espejismo del arreglo elemento a elemento. Tras la escritura, el contador R3 se incrementa en uno y el flujo regresa a la condición. Cuando R3 alcanza el valor 16 (los 16 elementos han sido copiados en orden inverso).



## Desarrollo

Listing 3: Código de la Actividad 3

```
1  /* ACTIVIDAD 3: Copia de arreglo invertida
2   Objetivo: A = [1..16], B = [16..1]
3 */
4 .data
5   A: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
6     @ Arreglo original de 16 datos
7   B: .skip 64           @ Arreglo vacío 'B' para la copia (64
8     bytes)
9
10
11 .text
12 .global main
13
14 main:
15   ldr r1, =A           @ R1 apunta al INICIO del arreglo
16     original 'A'
17   ldr r2, =B           @ R2 apunta al INICIO del arreglo
18     destino 'B'
19   add r2, r2, #60      @ Movemos R2 para que apunte al ÚLTIMO
20     espacio de 'B' (15 posiciones * 4 bytes = +60)
21   mov r3, #0            @ R3 es el contador, inicia en 0
22
23 loop_copia:
24   cmp r3, #16          @ ¿Ya copiamos 16 elementos?
25   beq fin_copia        @ Si sí, termina el ciclo
26
27   ldr r4, [r1], #4      @ Lee el dato apuntado por R1, lo
28     guarda en R4 y avanza R1 hacia ADELANTE (+4 bytes)
29   str r4, [r2], #-4      @ Escribe R4 en la dirección R2, y
30     mueve R2 hacia ATRÁS (-4 bytes)
31
32   add r3, r3, #1        @ Aumenta el contador de copiados
33   b loop_copia          @ Repite el ciclo
```



```
27 fin_copia:  
28     MOV R7, #1           @ sys_exit  
29     SVC 0               @ Termina programa
```

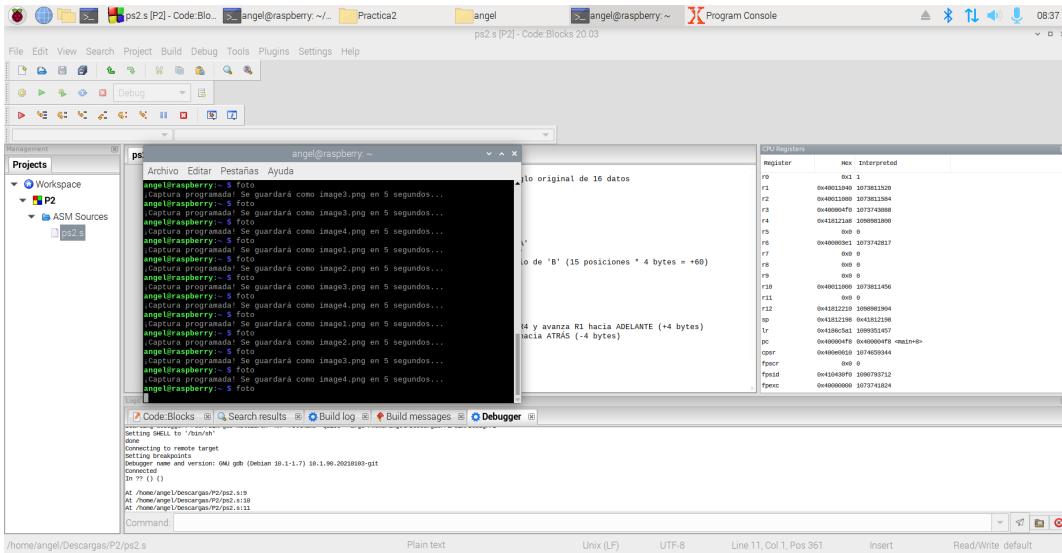


Figura 9: Entorno Code::Blocks preparado para iniciar la depuración del código fuente.

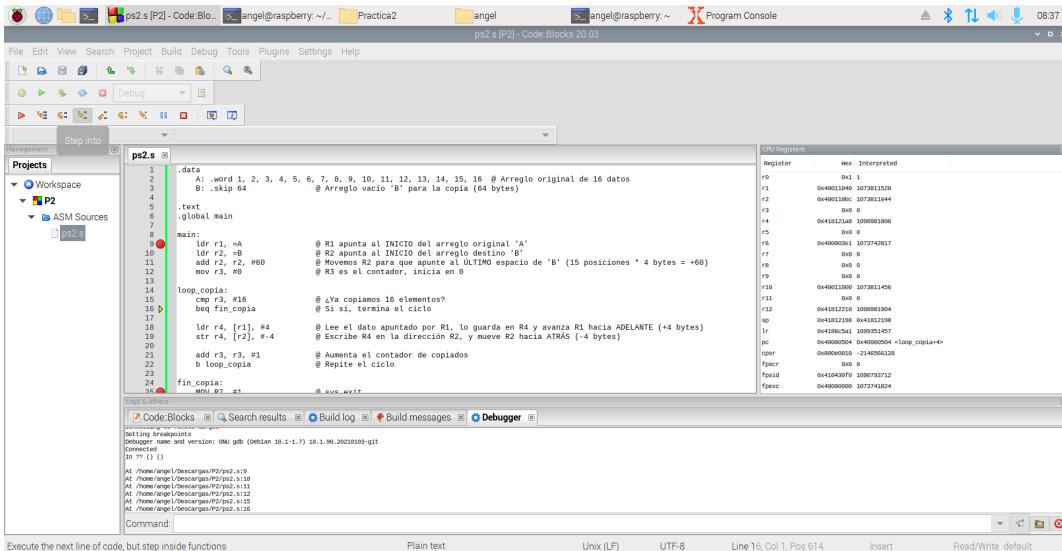


Figura 10: Inicialización de apuntadores. Se observa que R1 apunta al inicio de A (0x40011040) y R2 apunta al final de B (0x400110bc). El contador R3 inicia en 0.



The screenshot shows the Code::Blocks IDE interface with the following details:

- Title Bar:** ps2.s [P2] - Code.Blo... > angel@raspberry:~/... Practica2 > angel > angel@raspberry:~ - X Program Console
- Menu Bar:** File, Edit, View, Search, Project, Build, Debug, Tools, Plugins, Settings, Help
- Toolbar:** Includes icons for Run, Stop, Break, and Debug.
- Project Explorer:** Shows a workspace named "Workspace" containing a project "P2" which includes an "ASM Sources" folder with "ps2.s".
- Code Editor:** Displays the assembly code for "ps2.s". The code initializes a 16-word array "A" (0x40000000-0x4000001F), creates an empty array "B" (0x40000020-0x4000003F), and copies the contents of "A" to "B" using a loop. It then prints the copied data back to memory starting at R1.
- Registers:** Shows the CPU registers for the assembly code. The table includes columns for Register, Hex, and Interpreted values.
- Code Block:** Shows search results, build log, messages, and debugger tabs.
- Bottom Status Bar:** Includes tabs for Plain text, Unix (LF), UTF-8, Line 18, Col 1, Pos 672, Insert, and Read/Write default.

Figura 11: Evaluación de la condición de salida (CMP r3, #16) en las primeras fases del ciclo. La ejecución entra al bloque de copiado.

The screenshot shows the Code::Blocks IDE interface with the following details:

- File Menu:** File, Edit, View, Search, Project, Build, Debug, Tools, Plugins, Settings, Help.
- Toolbar:** Includes icons for Open, Save, Build, Run, Stop, and others.
- Project Explorer:** Shows "Workspace" and "P2" selected under "AS Sources".
- Code Editor:** Displays assembly code for "ps2.s".

```
ps2.s
1 .data
2 A: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16    @ Arreglo original de 16 datos
3 B: .skip 64          @ Arreglo vacío 'B' para la copia (64 bytes)
4
5 .text
6 global main
7
8 main:
9     ldr r1, =A          @ R1 apunta al INICIO del arreglo original 'A'
10    ldr r2, =B          @ R2 apunta al INICIO del arreglo destino 'B'
11    add r2, r2, #60      @ Movemos R2 para que apunte al ULTIMO espacio de 'B' (15 posiciones * 4 bytes = +60)
12    mov r3, #0           @ R3 es el contador, inicia en 0
13
14    loop_copia:
15        cmp r3, #16      @ ¿Ya copiamos 16 elementos?
16        beq fin_copia    @ Sí sí, termina el ciclo
17
18        ldr r4, [r1], #4    @ Lee el dato apuntado por R1, lo guarda en R4 y avanza R1 hacia ADELANTE (-4 bytes)
19        str r4, [r2], #4    @ Escribe R4 en la dirección R2, y mueve R2 hacia ATRÁS (-4 bytes)
20
21    ldi add r3, r3, #1    @ Aumenta el contador de copiados
22    b loop_copia         @ Repite el ciclo
23
24    fin_copia:
25        MOV.w r7, #1
26        B .here_exit
```
- CPU Registers Window:** Shows register values for R9 through R12, SP, PC, and various floating-point registers (FPR0-FPR12).

Register	Hex	Interpreted
R9	0x00000000	0x0 1
R2	0x400111000	1073712326
R2	0x400111000	1073712326
R3	0x0 3	
R4	0x4 4	
R5	0x0 0	
R6	0x40000001	1073742817
R7	0x0 0	
R8	0x0 0	
SP	0x0 0	
PC	0x400112000	10737121456
FPR0	0x418122100	1088981904
R11	0x0 0	
R12	0x418122100	1088981904
SP	0x418122100	1088981904
PC	0x418122100	1088981904
FPR0	0x400000010	0x400000010 <loop_copia+16>
FPR0	0x00000010	2140566128
FPR0	0x0 0	
FPR12	0x419043000	1089703712
FPR0	0x400000000	1073741824
- Log and Errors:** Shows build log and search results.
- Bottom Bar:** Execute the next line of code, Command: , Plain text, Unix (LF), UTF-8, Line 21, Col 1, Pos 887, Insert, Read/Write default.

Figura 12: Cuarta iteración. El contador R3 tiene el valor de 3, y en R4 se puede observar cargado el valor 0x4, demostrando que los apuntadores R1 y R2 se han actualizado correctamente.

Figura 13: Octava iteración del ciclo. El contador R3 llega a 7 y en R4 se carga el valor 0x7, confirmando la constancia y estabilidad del bucle.

Figura 14: Fin de la ejecución. El contador R3 alcanza el valor de 0x10 (16 en decimal). El programa sale del ciclo y ejecuta la llamada al sistema (SVC 0).

## Análisis de resultados

En una primera instancia, antes de ingresar al ciclo principal, es necesario cargar las direcciones base de los arreglos definidos en memoria. Por medio de la instrucción LDR, se

17



---

carga en el registro R1 la dirección de inicio del arreglo original A, la cual corresponde al valor 0x40011040. De la misma forma, se carga en R2 la dirección base del arreglo vacío B (0x40011080). Sin embargo, dado que la copia debe realizarse en sentido inverso, se emplea la instrucción ADD R2, R2, #60 para desplazar el apuntador de escritura hacia el final del espacio reservado para B. Como cada uno de los 16 datos es de 32 bits (4 bytes), el desplazamiento total es de 60 bytes, lo que posiciona correctamente a R2 en la dirección 0x400110BC. Adicionalmente, se inicializa el registro R3 con el valor de 0 para fungir como la variable de control (contador) del ciclo. Todos estos valores iniciales coinciden exactamente con lo que se muestra en los registros de la CPU en las primeras etapas de la depuración.

Una vez dentro de la etiqueta `loop_copia`, el programa ejecuta la lógica central del copiado haciendo uso del direccionamiento indirecto con post-indexado. La instrucción LDR R4, [R1], #4 accede a la dirección de memoria que contiene R1, extrae el dato y lo guarda en el registro temporal R4; inmediatamente después de la lectura, el procesador incrementa automáticamente el valor de R1 en 4 bytes para apuntar al siguiente dato del arreglo A. Posteriormente, la instrucción STR R4, [R2], #-4 toma el dato recién cargado en R4 y lo guarda en la dirección de memoria apuntada por R2; una vez almacenado, el apuntador R2 se decrementa automáticamente en 4 bytes. Este emparejamiento de instrucciones permite leer el arreglo original de inicio a fin mientras se escribe simultáneamente en el arreglo destino de fin a inicio, sin necesidad de emplear instrucciones aritméticas extra para recalcular las direcciones.

Al analizar la evolución dinámica a través de las iteraciones capturadas, se verifica que los datos se transfieren correctamente. Por ejemplo, en las primeras iteraciones se observa que R4 adquiere los valores de 0x1 y posteriormente 0x4, reflejando la extracción secuencial de los datos. De forma concurrente, el apuntador de lectura R1 incrementa progresivamente (pasando por 0x40011044, 0x40011050, hasta 0x4001105C), mientras que el apuntador de escritura R2 decrementa su valor (pasando por 0x400110B8, 0x400110AC, hasta 0x400110A0). En cada vuelta, la instrucción ADD R3, R3, #1 incrementa el contador, lo que permite llevar el control exacto de los elementos transferidos.

Finalmente, el ciclo se rompe gracias a la instrucción de comparación CMP R3, #16. Cuando el contador R3 alcanza el valor de 0x10 (16 en decimal), la comparación resulta en cero, lo que actualiza el registro de estado (CPSR) levantando la bandera de cero (Z). Al detectarse esta bandera, se cumple la condición de la instrucción BEQ `fin_copia`, realizando el salto fuera



---

del bucle. Al finalizar el programa, los registros muestran que R1 terminó en la dirección 0x40011080 (habiendo recorrido exactamente los 64 bytes del arreglo A) y R2 terminó en 0x4001107C (habiendo retrocedido 64 bytes desde su punto de inicio).

## Actividad 4

Realizar un programa que forme un arreglo de 20 elementos, con el siguiente criterio:

$$A = [i, 2i, 4i, 8i, 16i, \dots, ni]$$

Donde  $i$  es un número considerado como valor inicial.

- a) Enviar a memoria cada uno de ellos.
- b) Sumar y almacenar en memoria el resultado.

## Propuesta de solución

Para realizar un programa que resuelva la actividad planteada, primero se debe conocer el espacio que es necesario reservar para su correcto funcionamiento, de forma que el enunciado requiere un arreglo  $A$  de 20, elementos, dado que cada entero (**word**) ocupa 4 **bytes** es necesario reservar 80 bytes consecutivos para el arreglo de 20 elementos (A). Por otro lado se requiere una variable para almacenar el resultado, de forma que se reserva un espacio de 4 **bytes** y se inicializa en cero (**SUMA**).

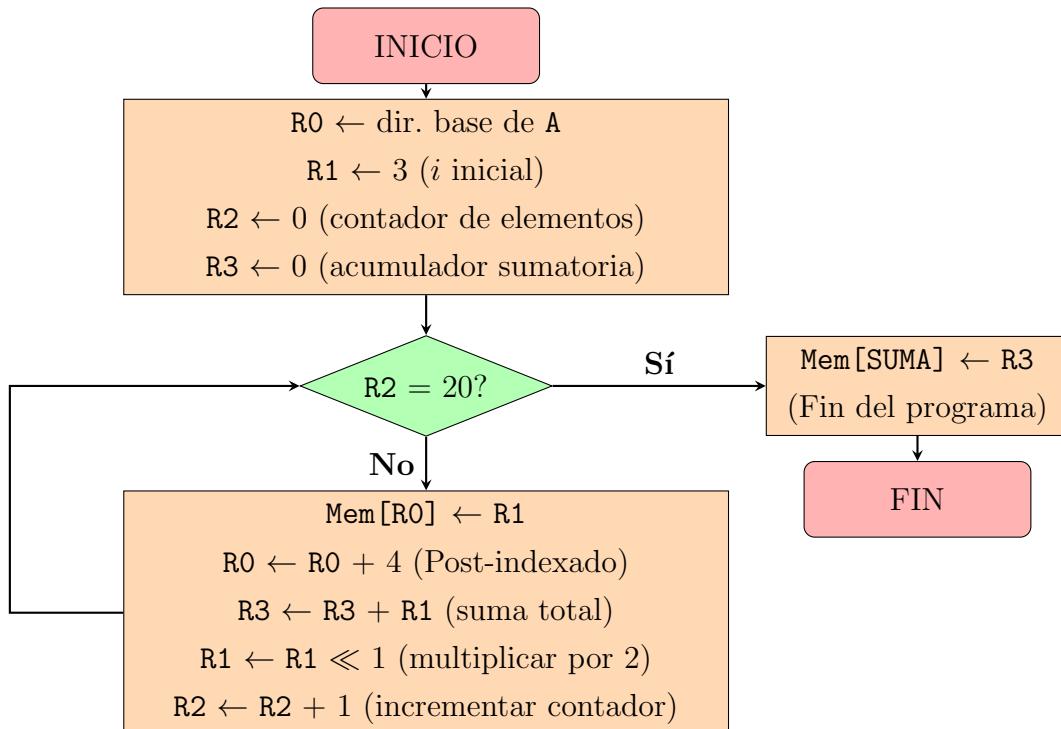
Una vez reservado el espacio es necesario cargar las direcciones de memoria para el correcto funcionamiento del programa, de forma que es necesario cargar la dirección de memoria donde empieza el arreglo A, esta dirección de memoria se guardará en el registro R0, posteriormente en el registro R1 se indicara el valor de  $i$ , en este caso le asignamos el valor de 3, por otro lado en el registro R2 se tendrá como contenido el valor del contador necesario para recorrer todo el arreglo, es decir ira de 0 a 20 y por último en el registro R3 se tendrá el contenido del acumulador, el resultado de ir sumando cada elemento generado.

Una vez con las direcciones de memoria e iniciado el contenido de los registros para el funcionamiento del programa, se ejecuta la etiqueta **loop\_potencias** donde se evalúa la condición de salida, la cál verifica si el contador ya llego a los 20 elementos (**CMP R2, #20**) en caso de

que no haya llegado al último elemento se procede a ir recorriendo el arreglo A, para esto guardamos el valor del registro R1 en la dirección apuntada por R0 (Arreglo A) y a su vez realizamos un post indexado sumándole 4, es mediante este post indexado (`str r1, [r0], #4`) que en cada ciclo recorremos el arreglo, se le suma 4 debido a que cada elemento del arreglo ocupa 4 bytes. Una vez que ya guardamos el valor, procedemos a sumar el valor recién guardado a la variable acumuladora que se encuentra en R3 (`add r3, r3, r1`). Por último es necesario generar la serie con la forma  $i, 2i, 4i, 8i, \dots$  para obtener esta forma se observó que en cada iteración se multiplica la constante por dos, de forma que se puede implementar por medio de un desplazamiento lógico a la izquierda, por lo que en cada ciclo incrementa la constante en 2, una vez finalizado esta secuencia de pasos se incrementa el contador R2 y se repite el ciclo evaluando la condición de salida.

Cuando la condición de salida es verdadera, en el contenido del registro R3 se tiene resultado de almacenar en memoria el resultado del arreglo, sin embargo la actividad indica que se debe almacenar el resultado, de forma que apuntamos a la dirección de memoria de SUMA (`ldr r0, =SUMA`), posteriormente guardamos el total acumulado en esa dirección de memoria (`str r3, [r0]`) finalmente se llama al sistema para finalizar el programa de forma segura.

El procedimiento descrito se puede representar en el siguiente diagrama de flujo.





## Desarrollo

Listing 4: Código de la Actividad 4

```
1  /* ACTIVIDAD 4: Arreglo exponencial y su suma
2   Objetivo: Generar serie multiplicando por 2 (Shift), y sumar
3   elementos.
4 */
5 .data
6   A:      .skip 80           @ Reserva memoria para 20 elementos
7   (20 * 4 bytes = 80)
8   SUMA:   .word 0           @ Variable para guardar la sumatoria
9   final
10
11 .text
12 .global main
13
14 main:
15   ldr r0, =A               @ R0 apunta a la dirección de memoria
16   de A
17   mov r1, #3                @ R1 será la variable 'i' inicial (
18   Ejemplo: usamos 3)
19   mov r2, #0                @ R2 es el contador de elementos
20   creados
21   mov r3, #0                @ R3 será el Acumulador (Sumatoria),
22   inicia en 0
23
24 loop_potencias:
25   cmp r2, #20              @ Compara si ya generamos los 20
26   elementos
27   beq fin_potencias        @ Si llegamos a 20, salimos del bucle
28
29   str r1, [r0], #4          @ Guarda el valor actual en memoria y
30   avanza el puntero R0
31   add r3, r3, r1            @ Suma el valor actual de 'i' al
32   Acumulador Total (R3)
33   lsl r1, r1, #1            @ Desplazamiento Izquierdo: Multiplica
```

```

'i' por 2 para la siguiente iteración
24   add r2, r2, #1           @ Incrementa contador
25   b loop_potencias         @ Repite

26
27 fin_potencias:
28   ldr r0, =SUMA            @ Carga la dirección de la variable
29   SUMA
30   str r3, [r0]              @ Guarda el resultado total (R3) en
31   esa memoria
32   MOV R7, #1                @ sys_exit
33   SVC 0                     @ Termina

```

The screenshot shows the Code::Blocks IDE interface. The left pane displays the assembly code for the file `ps2.s`. The right pane shows the CPU Registers window, which lists the values of various registers (R0-R12, SP, LR, PC, CPSR, FPCSR, FPSID, FPEXC) in hex and interpreted formats. The assembly code includes comments explaining the purpose of each instruction, such as reserving memory for array A, initializing pointers, and performing the loop logic.

Register	Hex	Interpreted
R0	0x40011040	1073811520
R1	0x3	3
R2	0x0	0
R3	0x40000440	1073743864
R4	0x41812180	1098981860
R5	0x0	0
R6	0x40000001	1073742817
R7	0x0	0
R8	0x0	0
R9	0x0	0
R10	0x40011000	1073811456
R11	0x0	0
R12	0x41812210	1098981964
SP	0x41812180	1098981218
LR	0x4186c5d1	1099351547
PC	0x400000fc	0x400000fc <main+12>
CPSR	0x40000000	1074038344
FPCSR	0x0	0
FPSID	0x410480f0	1096793712
FPEXC	0x40000000	1073741824

Figura 15: Estado de los registros. Dirección de memoria inicial del arreglo A



The screenshot shows the Code::Blocks IDE interface. The assembly code in the main window is:

```
.global main
main:
    ldr r0, =A          @ R0 apunta a la dirección de memoria de A
    mov r1, #3          @ R1 será la variable 'i' inicial (Ejemplo: usamos 3)
    mov r2, #0          @ R2 es el contador de elementos creados
    mov r3, #0          @ R3 será el Acumulador (Sumatoria), inicia en 0

loop_potencias:
    cmp r2, #20         @ Compara si ya generamos los 20 elementos
    beq fin_potencias  @ Si llegamos a 20, salimos del bucle
    str r1, [r0], #4    @ Guarda el valor actual en memoria y avanza el puntero R0
    add r3, r3, r1      @ Suma el valor actual de 'i' al Acumulador Total (R3)
    lsl r1, r1, #1      @ Desplazamiento Izquierdo: Multiplica 'i' por 2 para la siguiente iteración
    add r2, r2, #1      @ Incrementa contador
    b loop_potencias   @ Repite

fin_potencias:
    ldr r0, =SUMA       @ Carga la dirección de la variable SUMA
    str r3, [r0]         @ Guarda el resultado total (R3) en esa memoria
    MOV R7, #1           @ sys_exit
    SVC 0               @ Termina
```

The CPU Registers window shows the initial state of registers:

Register	Hex	Interpreted
r0	0x40011044	1073811524
r1	0x3	
r2	0x0	
r3	0x0	
r4	0x41812108	1099801800
r5	0x0	
r6	0x40000003	1073742817
r7	0x0	
r8	0x0	
r9	0x0	
r10	0x40011000	1073811456
r11	0x0	
r12	0x41812100	1099801804
sp	0x41812100	1099793712
lr	0x4186c541	1099351457
pc	0x40000000	<loop_potencias+12>
cpsr	0x00000010	-214056128
fpscr	0x0	
fpsid	0x410430f0	1099793712
fpexc	0x00000000	1073741824

Figura 16: Estado de los registros. Inicio de la etiqueta `loop_potencias`

The screenshot shows the Code::Blocks IDE interface. The assembly code in the main window is identical to Figure 16.

The CPU Registers window shows the state after the third iteration:

Register	Hex	Interpreted
r0	0x40011044	1073811528
r1	0x2	
r2	0x9	
r3	0x0	
r4	0x41812108	1099801800
r5	0x0	
r6	0x40000003	1073742817
r7	0x0	
r8	0x0	
r9	0x0	
r10	0x40011000	1073811456
r11	0x0	
r12	0x41812100	1099801804
sp	0x41812100	1099793712
lr	0x4186c541	1099351457
pc	0x40000000	<loop_potencias+24>
cpsr	0x00000010	-214056128
fpscr	0x0	
fpsid	0x410430f0	1099793712
fpexc	0x00000000	1073741824

Figura 17: Estado de los registros. Iteración número 3 de `loop_potencias`

The screenshot shows the Code::Blocks IDE interface. On the left, the Projects panel displays a workspace named 'P2' containing an 'ASM Sources' folder with 'ps2.s'. The main window shows the assembly code for 'ps2.s' with comments explaining the logic. The right side of the interface shows the 'CPU Registers' window, which lists various registers (R0-R15) with their memory addresses and values. Below the assembly code, the 'Build Log' window shows the compilation process for 'ps2.s'.

```

ps2.s
.global main
main:
    ldr r0, =A          // R0 apunta a la dirección de memoria de A
    mov r1, #3           // R1 es el valor de i (usamos 3)
    mov r2, #0           // R2 es el contador de elementos creados
    mov r3, #0           // R3 será el Acumulador (Sumatoria), inicia en 0

loop_potencias:
    cmp r2, #20         // Compara si ya generamos los 20 elementos
    beq fin_potencias   // Si llegamos a 20, salimos del bucle

    str r1, [r0], #4     // Guarda el valor actual en memoria y avanza el puntero R0
    add r2, r2, #1       // Suma 1 al valor actual en el Acumulador Total (R2)
    lsl r1, r1, #1       // Desplazamiento Izquierdo: Multiplica 'i' por 2 para la siguiente iteración
    add r2, r2, #1       // Incrementa contador
    b loop_potencias    // Repite

fin_potencias:
    ldr r0, =SUMA        // Carga la dirección de la variable SUMA
    str r3, [r0]          // Guarda el resultado total (R3) en esa memoria
    MOV R7, #1            // sys_exit
    SVC 0                // Termina

```

Register	Hex	Interpreted
R0	0x40011000	1073711600
R1	0x00000000	3145728
R2	0x14	20
R3	0xfffffd	3145725
R4	0x41011000	1073710300
R5	0x00000000	0
R6	0x40000001	1073742817
R7	0x1	1
R8	0x0	0
R9	0x0	0
R10	0x40011000	1073711406
R11	0x00000000	0
R12	0x41011000	1073710304
R0	0x41011000	1073710304
R1	0x41011000	1073710304
PC	0x40000002	0x4000002 <fin_potencias+2>
SP	0x00000010	1073713026
FPCR	0x00000000	0
FPSR	0x41000000	1073713712
FPCN	0x40000000	1073714124

Figura 18: Estado de los registros. Fin de Iteraciones, resultado final

## Análisis de resultados

A partir de las capturas de pantalla podemos observar paso a paso el funcionamiento del programa que acabamos de implementar, de forma que en la primer captura podemos observar que en el registro R0 se tiene la dirección de memoria inicial del arreglo A, así como en R1 es valor de  $i$  que especificamos 3 así como la variable contadora R2 en 0.

En la imagen que corresponde al inicio de la etiqueta `loop_potencias` se puede comprobar como se evalúa si la variable contadora R2 llegó al valor de 20 en caso que sea falso entra en el loop procediendo a realizar las operaciones dentro del ciclo. En la imagen posterior se puede corroborar el correcto funcionamiento de cada iteración, como estamos en la iteración número 3, vemos que en el R1 se tiene el contenido de 12 lo cual corresponde al resultado correcto de esa iteración  $4 \times 3$  de forma que hasta este momento en memoria se tendría en el contenido del arreglo

$$A = [i, 2i, 4i]$$

$$A = [3, 2 \times 3, 4 \times 3] = [3, 6, 12]$$

En la imagen donde se muestra el fin del ciclo, se puede observar que la variable contadora R2 llegó a 20 de forma que la condición se ejecuta de forma correcta, de igual forma se puede observar que el resultado final en R1 es correcta coincide con la sucesión planteada y se almacena en memoria el resultado final.



## Actividad 5

Realizar un programa que multiplique dos matrices de  $2 \times 2$ ; los datos podrán ser de 8 bits.

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} I & J \\ K & L \end{bmatrix}$$

### Propuesta de solución

Para la solución de la multiplicación de matriz de tamaño  $2 \times 2$ , se definen tres arreglos con 4 elementos de 8 bits, para esto se considera que un arreglo almacena los contenidos de una matriz de la siguiente forma  $M = [a_1, a_2, a_3, a_4]$  por lo que se considera que los datos se almacenan de forma continua. Una vez mencionando esto se indica que cada elemento es de 8 bits por medio de la `.byte` ya que se indica que cada numero solo ocupa 8 bits (`1 byte`) de forma que `M1` y `M2` contiene los valores de las matrices de entrada y `MR` es la matriz resultante de realizar la multiplicación entre ambas matrices por lo que se inicializa con 0 los elementos.

Una vez que ya se reservo la memoria de cada arreglo correspondiente a las matrices, es necesario guardar las direcciones de memoria de cada matriz, de forma que en el registro `R0` se tiene la dirección inicial del arreglo de correspondiente a la matriz `M1`, `R1` para la matriz `M2` y `R3` para la resultante `MR`. Una vez ya con los valores iniciales de la dirección de memoria, en los registros (`R3, R4, R5, R6`) se procede a cargar cada elemento de la matriz `M1`, los elementos se cargan por medio de la instrucción `LDRB`, el cual extrae únicamente 8 bits, ya que los elementos de cada matriz es de `8 bits`, para desplazarnos en el arreglo se emplea la sintaxis de `LDRB Rd, [Rn, #offset]` donde al registro que contiene la dirección de inicio del arreglo (`Rd`) se le indica cuantos bytes moverse (`#offset`) para encontrar el dato solicitado. Se emplean los registros (`R7, R8, R9, R10`) para los elementos de la matriz `M2` con la misma lógica descrita

Dado que se tienen matrices de tamaño fijo, se pueden calcular los elementos de la matriz por medio de las siguientes expresiones:

$$I = (A \times E) + (B \times G)$$

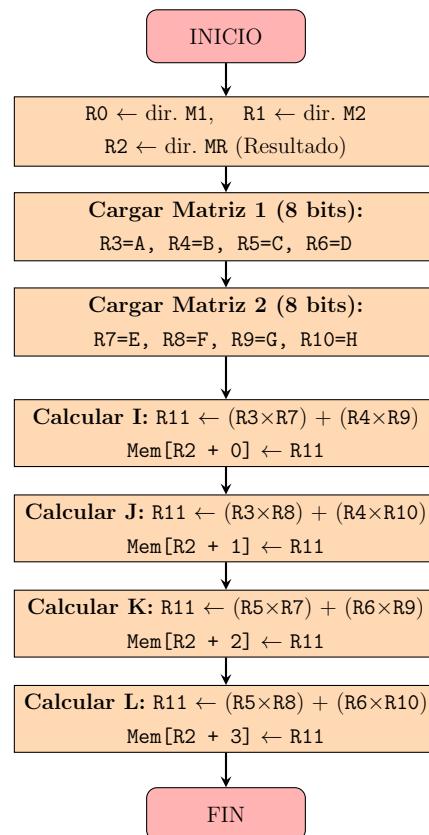
$$J = (A \times F) + (B \times H)$$

$$K = (C \times E) + (D \times G)$$

$$L = (C \times F) + (D \times H)$$

Una vez que se conocen las expresiones para calcular cada elemento de la matriz resultante se pueden escribir en lenguaje ensamblador, para esto se tienen que realizar 3 operaciones, 2 multiplicaciones y la suma ambas multiplicaciones, por lo que para optimizar el código se hace uso de las instrucciones **MLA** y **MUL**. Por ejemplo considerando el cálculo del elemento *I* se emplearía el siguiente procedimiento. Multiplicar  $A \times E$  y guardarlo en el registro R11 (**mul r11, r3, r7**), posteriormente multiplicamos  $B \times G$  y le sumamos el resultado de multiplicar  $A \times E$  y sobre escribimos el resultado, para hacer todo esto por medio de una instrucción se emplea la instrucción **MLA** (**mla r11, r4, r9, r11**). De forma que ya en R11 tenemos el resultado final del elemento por lo que solo queda guardar ese byte en la posición correspondiente de **MR** (**strb r11, [r2, #0]**). Esta lógica se emplea para cada elemento de la matriz con los respectivos registros que contienen los elementos de cada matriz.

Una vez que ya se termino de realizar las operaciones necesarias para calcular cada elemento de la matriz resultante se procede a ejecutar la llamada para terminar el proceso.



## Desarrollo

Listing 5: Código de la Actividad 5

```

1  /* ACTIVIDAD 5: Multiplicación de matrices 2x2
2      Objetivo: [A B] x [E F] = [I J]
3                  [C D]      [G H]      [K L]
4 */
5 .data
6     M1: .byte 2, 1, 3, 4          @ Matriz 1 (A,B,C,D) -> Valores de
7             ejemplo de 8 bits
8     M2: .byte 1, 5, 2, 1          @ Matriz 2 (E,F,G,H) -> Valores de
9             ejemplo de 8 bits
10    MR: .byte 0, 0, 0, 0         @ Matriz Resultado (I,J,K,L)
11
12
13 main:
14     ldr r0, =M1                @ Dirección Matriz 1
15     ldr r1, =M2                @ Dirección Matriz 2
16     ldr r2, =MR                @ Dirección Matriz Resultado
17
18     @ Cargamos los elementos de M1 (Usamos LDRB por ser Bytes)
19     ldrb r3, [r0, #0]           @ R3 = A (Posición 0)
20     ldrb r4, [r0, #1]           @ R4 = B (Posición 1)
21     ldrb r5, [r0, #2]           @ R5 = C (Posición 2)
22     ldrb r6, [r0, #3]           @ R6 = D (Posición 3)
23
24     @ Cargamos los elementos de M2
25     ldrb r7, [r1, #0]           @ R7 = E
26     ldrb r8, [r1, #1]           @ R8 = F
27     ldrb r9, [r1, #2]           @ R9 = G
28     ldrb r10,[r1, #3]          @ R10 = H
29
30     @ Calculando I = A*E + B*G
31     mul r11, r3, r7            @ R11 = A * E

```



```
32     mla r11, r4, r9, r11          @ Multiply-Accumulate: R11 = (B * G) +
      R11
33     strb r11, [r2, #0]           @ Guardamos 'I' en la matriz resultado
34
35     @ Calculando J = A*F + B*H
36     mul r11, r3, r8              @ R11 = A * F
37     mla r11, r4, r10, r11        @ R11 = (B * H) + R11
38     strb r11, [r2, #1]           @ Guardamos 'J'
39
40     @ Calculando K = C*E + D*G
41     mul r11, r5, r7              @ R11 = C * E
42     mla r11, r6, r9, r11        @ R11 = (D * G) + R11
43     strb r11, [r2, #2]           @ Guardamos 'K'
44
45     @ Calculando L = C*F + D*H
46     mul r11, r5, r8              @ R11 = C * F
47     mla r11, r6, r10, r11        @ R11 = (D * H) + R11
48     strb r11, [r2, #3]           @ Guardamos 'L'
49
50     MOV R7, #1                  @ sys_exit
51     SVC 0                      @ Termina
```



The screenshot shows the Code::Blocks IDE interface with the following details:

- Project:** ps2.s [P2] - Code: Blo...
- File:** ps2.s
- Registers:** CPU Registers table showing register values in Hex and Interpreted format.
- Memory:** RAM View showing memory dump and registers.
- Registers:** CPU Registers table showing register values in Hex and Interpreted format.
- Logs & others:** Build messages and debugger logs.
- Console:** Program Console showing the assembly code and its execution.

The assembly code in the editor is:

```
1 .data
2     M1:.byte 2, 1, 3, 4      @ Matriz 1 (A,B,C,D) -> Valores de ejemplo de 8 bits
3     M2:.byte 1, 5, 2, 1      @ Matriz 2 (E,F,G,H) -> Valores de ejemplo de 8 bits
4     MR:.byte 0, 0, 0, 0      @ Matriz Resultado (I,J,K,L)
5
6 .text
7     .global main
8
9 main:
10    ldr r0, =M1            @ Dirección Matriz 1
11    ldr r1, =M2            @ Dirección Matriz 2
12    ldr r2, =MR            @ Dirección Matriz Resultado
13
14    @ Cargamos los elementos de M1 (Usamos LDRB por ser Bytes)
15    ldrb r3, [r0, #0]       @ R3 = A (Posición 0)
16    ldrb r4, [r0, #1]       @ R4 = B (Posición 1)
17    ldrb r5, [r0, #2]       @ R5 = C (Posición 2)
18    ldrb r6, [r0, #3]       @ R6 = D (Posición 3)
19
20    @ Cargamos los elementos de M2
21    ldrb r7, [r1, #0]       @ R7 = E
22    ldrb r8, [r1, #1]       @ R8 = F
23    ldrb r9, [r1, #2]       @ R9 = G
24    ldrb r10,[r1, #3]      @ R10 = H
```

Figura 19: Estado de los registros. Dirección de memoria inicial de las matrices M1,M2, MR

The screenshot shows the Code::Blocks IDE interface with the following details:

- Project:** ps2.s [P2] - Code: Blo...
- File:** ps2.s
- Registers:** CPU Registers table showing register values in Hex and Interpreted format.
- Memory:** RAM View showing memory dump and registers.
- Registers:** CPU Registers table showing register values in Hex and Interpreted format.
- Logs & others:** Build messages and debugger logs.
- Console:** Program Console showing the assembly code and its execution.

The assembly code in the editor is identical to Figure 19:

```
1 .data
2     M1:.byte 2, 1, 3, 4      @ Matriz 1 (A,B,C,D) -> Valores de ejemplo de 8 bits
3     M2:.byte 1, 5, 2, 1      @ Matriz 2 (E,F,G,H) -> Valores de ejemplo de 8 bits
4     MR:.byte 0, 0, 0, 0      @ Matriz Resultado (I,J,K,L)
5
6 .text
7     .global main
8
9 main:
10    ldr r0, =M1            @ Dirección Matriz 1
11    ldr r1, =M2            @ Dirección Matriz 2
12    ldr r2, =MR            @ Dirección Matriz Resultado
13
14    @ Cargamos los elementos de M1 (Usamos LDRB por ser Bytes)
15    ldrb r3, [r0, #0]       @ R3 = A (Posición 0)
16    ldrb r4, [r0, #1]       @ R4 = B (Posición 1)
17    ldrb r5, [r0, #2]       @ R5 = C (Posición 2)
18    ldrb r6, [r0, #3]       @ R6 = D (Posición 3)
19
20    @ Cargamos los elementos de M2
21    ldrb r7, [r1, #0]       @ R7 = E
22    ldrb r8, [r1, #1]       @ R8 = F
23    ldrb r9, [r1, #2]       @ R9 = G
24    ldrb r10,[r1, #3]      @ R10 = H
```

Figura 20: Estado de los registros. Carga del contenido correspondiente a los elementos de M1



The screenshot shows the Code::Blocks IDE interface with the following details:

- Title Bar:** ps2.s [P2] - Code::Blo... angel@raspberry: ~ Práctica2 angel angel@raspberry: ~ Program Console
- File Menu:** File Edit View Search Project Build Debug Tools Plugins Settings Help
- Toolbar:** Includes icons for file operations, build, and debug.
- Project Explorer:** Projects workspace P2 ASM Sources ps2.s
- Code Editor:** Displays assembly code for calculating matrix products A'E + B'G, A'F + B'H, C'E + D'G, and C'F + D'H.
- Registers Window:** Shows CPU Registers for r0 to r12, sp, lr, pc, cpsr, fpSCR, fpSID, and fpExc.
- Logs & others:** Shows build log output.
- Command Line:** /home/angel/Desktop/P2/ps2.s
- Status Bar:** Plain text Unix (LF) UTF-8 Line 27, Col 1, Pos 923 Insert Read/Write default

Figura 21: Estado de los registros. Carga del contenido correspondiente a los elementos de M2

The screenshot shows the Code::Blocks IDE interface with the following details:

- Title Bar:** ps2.s [P2] - Code::Blo... angel@raspberry: ~ Práctica2 angel angel@raspberry: ~ Program Console
- File Menu:** File Edit View Search Project Build Debug Tools Plugins Settings Help
- Toolbar:** Includes icons for file operations, build, and debug.
- Project Explorer:** Projects workspace P2 ASM Sources ps2.s
- Code Editor:** Displays assembly code for calculating matrix products A'E + B'G, A'F + B'H, C'E + D'G, and C'F + D'H.
- Registers Window:** Shows CPU Registers for r0 to r12, sp, lr, pc, cpsr, fpSCR, fpSID, and fpExc.
- Logs & others:** Shows build log output.
- Command Line:** /home/angel/Desktop/P2/ps2.s
- Status Bar:** Plain text Unix (LF) UTF-8 Line 42, Col 1, Pos 1489 Insert Read/Write default

Figura 22: Estado de los registros. Calculo de los elementos correspondientes de la matriz MR

The screenshot shows the Code::Blocks IDE interface. On the left, the project tree shows 'ps2.s' under 'P2'. The main window displays assembly code for calculating matrix multiplication. The code includes comments explaining operations like 'Calculando I = A \* E + B \* G', 'Calculando J = A \* F + B \* H', 'Calculando K = C \* E + D \* G', and 'Calculando L = C \* F + D \* H'. The CPU Registers window on the right lists registers R0 through R13 with their memory addresses and values.

```

# Calculando I = A * E + B * G
25    mul r11, r3, r7      # R11 = A * E
26    mla r11, r4, r9, r11  # Multiply-Accumulate: R11 = (B * G) + R11
27    strb r11, [r2, #0]    # Guardamos 'I' en la matriz resultado
28
29
30
31
32    # Calculando J = A * F + B * H
33    mul r11, r5, r7      # R11 = A * F
34    mla r11, r4, r10, r11 # R11 = (B * H) + R11
35    strb r11, [r2, #1]    # Guardamos 'J'
36
37
38    # Calculando K = C * E + D * G
39    mul r11, r5, r7      # R11 = C * E
40    mla r11, r6, r10, r11 # R11 = (D * G) + R11
41    strb r11, [r2, #2]    # Guardamos 'K'
42
43
44    # Calculando L = C * F + D * H
45    mul r11, r5, r8      # R11 = C * F
46    mla r11, r6, r10, r11 # R11 = (D * H) + R11
47    strb r11, [r2, #3]    # Guardamos 'L'
48
49    MOV R7, #1           # sys_exit
50    SVC 8                # Termina

```

Register	Hex	Interpreted
R0	0x40011040	1073811230
R1	0x40011044	1073811234
R2	0x40011048	1073811238
R3	0x4001104C	1073811232
R4	0x40011050	1073811236
R5	0x40011054	1073811230
R6	0x40011058	1073811244
R7	0x4001105C	1073811241
R8	0x40011060	1073811245
R9	0x40011064	1073811242
R10	0x40011068	1073811241
R11	0x40011070	1073811249
R12	0x41400000	1073811204
R0	0x41400000	1073811204
R1	0x41400001	1073811205
PC	0x40000000	0x40000000 <main+0>
SP	0x40000010	1073810944
FP	0x0	0
FPC	0x41400000	1073813712
FPCN	0x40000000	1073811224

Figura 23: Estado de los registros. Fin de calculo multiplicación de las matrices M1 x M2

## Análisis de resultados

En la primer imagen de la depuración correspondiente a la actividad 6, se pude observar de manera correcta como se apunta para cada matriz a la dirección inicial de cada arreglo que corresponde a las matrices M1,M2,MR, dichas direcciones se pueden ver que están en el contenido de los registros R0,R1,R2.

Al cargar el contenido de los elementos de 8 bits que hay en esa dirección de memoria con los desplazamientos correspondientes para acceder a los 4 elementos que hay en ese arreglo, en los CPU\_Registers se puede observar que coinciden con los valores que se definieron al inicio del programa a la hora de reservar memoria. En la imagen siguiente ya se pueden observar los elementos correspondientes a la matriz M2 de forma que se puede afirmar que se reservó de forma correcta la memoria para cada una de las matrices, ya que se accedieron a esas direcciones de memoria y el contenido corresponde con los elementos de las matrices.

En la penúltima imagen se puede observar el resultado de calcular el elemento K de la matriz resultante, para verificar que el resultado es correcto se realizó el cálculo de las multiplicaciones para las matrices que se definieron:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} I & J \\ K & L \end{bmatrix}$$

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} (A \times E) + (B \times G) & (A \times F) + (B \times H) \\ (C \times E) + (D \times G) & (C \times F) + (D \times H) \end{bmatrix}$$

$$\begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 1 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} (2 \times 1) + (1 \times 2) & (2 \times 5) + (1 \times 1) \\ (3 \times 1) + (4 \times 2) & (3 \times 5) + (4 \times 1) \end{bmatrix}$$

$$\begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 1 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 11 \\ 11 & 19 \end{bmatrix}$$

De forma que el elemento K calculado corresponde con el contenido del registro R11, finalmente cuando acaba de realizarse la multiplicación de matrices el contenido del registro R11 debe corresponder al último elemento que se cálculo, este caso para la matriz de  $2 \times 2$  es el elemento L que coincide su contenido con el resultado esperado. Por lo que el programa realiza de forma correcta la multiplicación de matrices de tamaño  $2 \times 2$

## Actividad 6

Realizar un programa que encuentre el número con valor mayor en un arreglo de 20 elementos que serán almacenados en memoria; para lo cual:

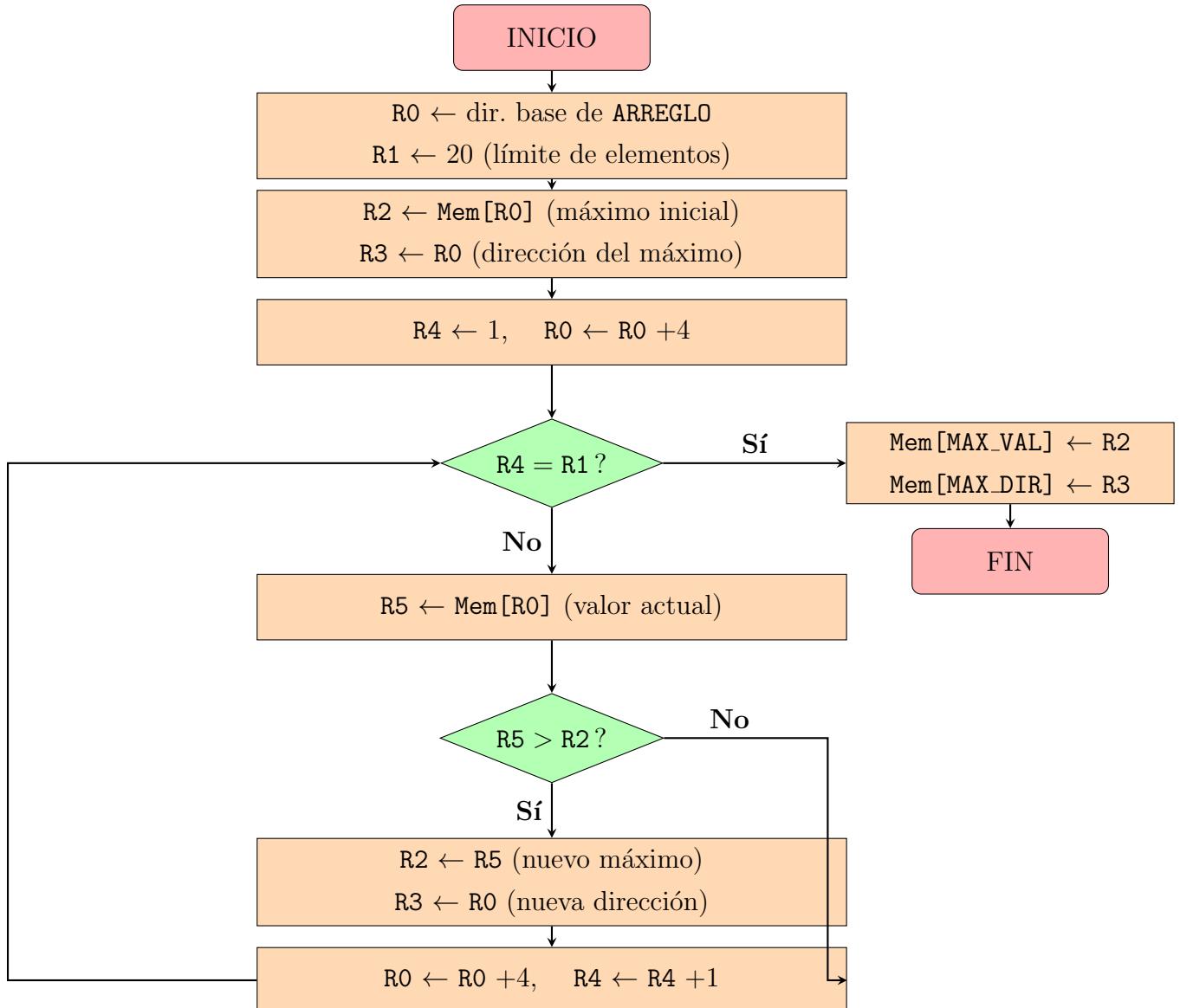
- a) Indicar cuál fue el valor mayor.
- b) Ubicar la dirección donde se encontró este número.
- c) Usar las direcciones que requiera para cumplir lo solicitado.

## Propuesta de solución

Se emplea un recorrido lineal sobre el arreglo de 20 elementos, comparando cada valor contra un máximo registrado. Se inicializa el registro R2 con el primer elemento del arreglo, asumiendo que este es el mayor, y R3 con su dirección de memoria correspondiente. A partir del segundo elemento, un ciclo iterativo controlado por el contador R4 recorre las posiciones restantes: en cada paso, la instrucción LDR carga el valor actual en R5 y la instrucción CMP lo compara contra el máximo almacenado en R2. Si el nuevo valor supera al registrado, las instrucciones MOV actualizan tanto el valor máximo como su dirección. Cuando el contador

R4 iguala al límite de 20, el programa sale del ciclo y almacena el resultado final en las variables de memoria MAX\_VAL y MAX\_DIR mediante instrucciones STR, cumpliendo así con los tres incisos solicitados.

A continuación se presenta el diagrama de flujo correspondiente al algoritmo descrito:



El diagrama inicia cargando la dirección base del arreglo en R0 y estableciendo el límite de 20



elementos en R1. Se asume que el primer dato es el mayor, por lo que R2 almacena su valor y R3 su dirección de memoria. El contador R4 se inicializa en 1 y el puntero R0 avanza al segundo elemento (+4 bytes). En cada iteración del ciclo principal se evalúa primero si el contador ha alcanzado el límite: cuando R4 = R1 (20), el flujo se dirige al bloque de almacenamiento donde se escriben los resultados en las variables MAX\_VAL y MAX\_DIR, finalizando el programa. Si la condición no se cumple, se carga el valor actual de memoria en R5 y se compara con el máximo registrado en R2. Si R5 resulta mayor, los registros R2 y R3 se actualizan con el nuevo valor y su dirección; en caso contrario, se omite la actualización saltando directamente al avance del puntero. Finalmente, se incrementa R0 en 4 bytes y R4 en una unidad, regresando el flujo al inicio del ciclo.

## Desarrollo

Listing 6: Código de la Actividad 6

```
1 /* ACTIVIDAD 6: Búsqueda del número mayor en arreglo
2     Objetivo: Encontrar el máximo y guardar su valor y su dirección
3     de memoria.
4 */
5 .data
6     @ Arreglo de 20 números al azar para la prueba
7     ARREGLO: .word 5, 12, 3, 45, 2, 105, 1, 8, 33, 10, 11, 14, 0,
8         77, 21, 6, 9, 88, 4, 15
9     MAX_VAL: .word 0           @ Variable para guardar el número más
10    grande
11    MAX_DIR: .word 0          @ Variable para guardar la dirección
12    de memoria de ese número
13
14 .text
15 .global main
16
17 main:
18     ldr r0, =ARREGLO          @ R0 = Puntero principal que recorrerá
19         el arreglo
20     mov r1, #20              @ R1 = Límite de elementos (20)
21     ldr r2, [r0]              @ R2 = Guarda el MÁXIMO (Inicia
```



```
        asumiendo que el índice 0 es el mayor)

17  mov r3, r0                      @ R3 = Guarda la DIRECCIÓN del máximo
   (Inicia con la del índice 0)

18  mov r4, #1                       @ R4 = Contador de ciclo (inicia en 1
   porque ya evaluamos el 0)

19  add r0, r0, #4                  @ Avanzamos el puntero de memoria al i-
   ndice 1

20

21 buscar_mayor:
22  cmp r4, r1                      @ Compara el contador con 20
23  beq fin_busqueda              @ Si terminamos, salta al final

24

25  ldr r5, [r0]                    @ R5 = Lee el valor actual de la
   memoria
26  cmp r5, r2                      @ Compara (Valor_Actual vs Má-
   ximo_Registrado)
27  bne siguiente                  @ Branch if Less or Equal: Si es menor
   o igual, ignóralo y salta a 'siguiente'

28  @ Si llegó a esta línea, encontramos un nuevo mayor
29  mov r2, r5                      @ R2 adopta el nuevo valor mayor
30  mov r3, r0                      @ R3 adopta la dirección de memoria de
   este nuevo mayor

31

32 siguiente:
33  add r0, r0, #4                @ Avanzamos la lectura en la memoria
   (4 bytes)
34  add r4, r4, #1                @ Incrementamos el contador de ciclo
35  b buscar_mayor               @ Repetimos

36

37 fin_busqueda:
38  ldr r6, =MAX_VAL             @ Carga dirección para guardar el
   valor
39  str r2, [r6]                  @ Almacena en memoria el valor mayor
40  ldr r6, =MAX_DIR              @ Carga dirección para guardar la
   ubicación
```



```
42     str r3, [r6]           @ Almacena en memoria la dirección del
        mayor

43

44     MOV R7, #1             @ sys_exit
45     SVC 0                  @ Terminar
```

The screenshot shows the Code::Blocks IDE interface. The top menu bar includes File, Edit, View, Search, Project, Build, Debug, Tools, Plugins, Settings, Help, and a Language selector set to English. The toolbar contains icons for file operations like Open, Save, and Print, as well as build-related tools like Build, Run, and Stop.

The left sidebar displays the Project Management window with a Projects tree showing a workspace named 'P2' containing an ASIM Sources folder with a file named 'ps2.s'. The main editor window shows the assembly code for the file:

```
ps2.s
1 .data
2     ARREGLO: .word 5, 12, 3, 45, 2, 105, 1, 6, 9, 10, 11, 14, 0, 77, 21, 6, 9, 88, 4, 15
3     MAX_VAL: .word 0           ; Variable para guardar el numero más grande
4     MAX_DIR: .word 0           ; Variable para guardar la dirección de memoria de ese número
5
6 .text
7     .global main
8
9 main:
10    ldr r0, =ARREGLO          ; R0 = Puntero principal que recorrerá el arreglo
11    mov r1, #20                ; R1 = Límite de elementos (20)
12    ldr r2, [r0]                ; R2 = Guarda el MAXIMO (Inicia sumiendo que el indice 0 es el mayor)
13    mov r3, #0
14    mov r4, #1
15    add r0, r0, #4             ; Avanzamos el puntero de memoria al indice 1
16    cmp r4, r1                ; Compara el contador con 20
17    beq fin_busqueda          ; Si termina, salta al final
18
19    ldr r5, [r0]                ; R5 = Lee el valor actual de la memoria
20    cmp r5, r2                ; Compara (Valor_Actual vs Maximo_Registrado)
21    bne siguiente             ; Branch if Less or Equal: Si es menor o igual, ignóralo y salta a 'siguiente'
22
23    fin_busqueda:
24
25
26
```

The right side of the interface shows the CPU Registers window, displaying the state of various registers (r0-r12, sp, pc, etc.) and memory locations at address 0x40000000.

Figura 24: Estado de los registros inmediatamente después de la inicialización de variables. Se asume que el índice 0 es el máximo.



The screenshot shows the Code::Blocks IDE interface. The assembly code for file ps2.s is displayed in the main window, showing a loop to find the maximum value in memory. The CPU Registers window shows the state of registers R0 through R13 at the end of iteration 1. The assembly code includes comments explaining the purpose of each instruction.

```
ps2.s
13 ldr r2, [r0]          @ R2 = Guarda el MAXIMO (Inicia asumiendo que el indice 0 es el mayor)
14 mov r3, r0            @ R3 = Guarda la DIRECCION del maximo (Inicia con la del indice 0)
15 mov r4, #1             @ R4 = Contador de ciclo (Inicia en 1 porque ya evaluamos el 0)
16 add r0, r0, #4          @ Avanzamos el puntero de memoria al indice 1
17
18 buscar_mayor:
19     cmp r4, r1          @ Compara el contador con 20
20     beq fin_busqueda    @ Si terminamos, salta al final
21
22     ldr r5, [r0]          @ R5 = Lee el valor actual de la memoria
23     cmp r5, r2            @ Compara (Valor_Actual vs Maximo_Registrado)
24     ble siguiente         @ Branch if Less or Equal: Si es menor o igual, ignóralo y salta a 'siguiente'
25
26     @ Si llego a esta linea, encontramos un nuevo mayor
27     mov r3, r5            @ R3 adopta la direccion de memoria de este nuevo mayor
28
29     siguiente:
30         add r0, r0, #4        @ Avanza la lectura en la memoria (4 bytes)
31         add r4, r4, #1        @ Incrementamos el contador de ciclo
32         b buscar_mayor       @ Repetimos
33
34 fin_busqueda:
35     ldr r6, =MAX_VAL      @ Carga direccion para guardar el valor
36     str r2, [r6]           @ Almacena en memoria el valor_mayor
37
```

Register	Hex	Interpreted
r0	0x40011040	1073811230
r1	0x40011044	1073811234
r2	0x40011048	1073811238
r3	0x4001104C	0
r4	0x40011050	2
r5	0x40011054	3
r6	0x40011058	4
r7	0x4001105C	1
r8	0x40011060	5
r9	0x40011064	2
r10	0x40011068	1
r11	0x4001106C	2
r12	0x41400000	1073811234
sp	0x41400100	0x41400100
lr	0x41400104	1073811238
pc	0x40000020	0x40000020 <buscar_mayor>
cpar	0x40000010	1073811234
fpscr	0x0	0
fpcsr	0x41400000	1073811232
fpcsc	0x40000000	1073811234

Figura 25: Interrupción dentro del bloque de actualización (`mov r3, r0`) durante la segunda iteración, al encontrar un número mayor que el inicial.

The screenshot shows the Code::Blocks IDE interface. The assembly code for file ps2.s is displayed in the main window, showing the loop to find the maximum value in memory. The CPU Registers window shows the state of registers R0 through R13 at the end of iteration 6. The assembly code includes comments explaining the purpose of each instruction.

```
ps2.s
13 ldr r2, [r0]          @ R2 = Guarda el MAXIMO (Inicia asumiendo que el indice 0 es el mayor)
14 mov r3, r0            @ R3 = Guarda la DIRECCION del maximo (Inicia con la del indice 0)
15 mov r4, #1             @ R4 = Contador de ciclo (Inicia en 1 porque ya evaluamos el 0)
16 add r0, r0, #4          @ Avanzamos el puntero de memoria al indice 1
17
18 buscar_mayor:
19     cmp r4, r1          @ Compara el contador con 20
20     beq fin_busqueda    @ Si terminamos, salta al final
21
22     ldr r5, [r0]          @ R5 = Lee el valor actual de la memoria
23     cmp r5, r2            @ Compara (Valor_Actual vs Maximo_Registrado)
24     ble siguiente         @ Branch if Less or Equal: Si es menor o igual, ignóralo y salta a 'siguiente'
25
26     @ Si llego a esta linea, encontramos un nuevo mayor
27     mov r3, r5            @ R3 adopta la direccion de memoria de este nuevo mayor
28
29     siguiente:
30         add r0, r0, #4        @ Avanza la lectura en la memoria (4 bytes)
31         add r4, r4, #1        @ Incrementamos el contador de ciclo
32         b buscar_mayor       @ Repetimos
33
34 fin_busqueda:
35     ldr r6, =MAX_VAL      @ Carga direccion para guardar el valor
36     str r2, [r6]           @ Almacena en memoria el valor_mayor
37
```

Register	Hex	Interpreted
r0	0x40011040	1073811234
r1	0x40011044	0
r2	0x40011048	1073811238
r3	0x4001104C	0x69
r4	0x40011050	6
r5	0x40011054	255
r6	0x40011058	0x41400000
r7	0x40011060	0
r8	0x40011064	0
r9	0x40011068	0
r10	0x4001106C	1073811238
r11	0x40011070	0
r12	0x41400000	1073811234
sp	0x41400100	0x41400100
lr	0x41400104	1073811238
pc	0x40000020	0x40000020 <buscar_mayor>
cpar	0x40000010	1073811234
fpscr	0x0	0
fpcsr	0x41400000	1073811232
fpcsc	0x40000000	1073811234

Figura 26: El ciclo cursando la iteración 6 (`R4 = 0x6`). El programa ya ha registrado el verdadero número máximo (`0x69`).

```

13 ldr r2, [r0]          @ R2 = Guarda el MAXIMO. (Inicia asumiendo que el indice 0 es el mayor)
14 mov r3, r0            @ R3 = Guarda la DIRECCIÓN del máximo (Inicia con la del índice 0)
15 mov r4, #1             @ R4 = Contador de ciclo (inicia en 1 porque ya evaluamos el 0)
16 add r0, r0, #4         @ Avanzamos el puntero de memoria al índice 1
17
18 buscar_mayor:
19     cmp r4, r1          @ Compara el contador con 20
20     beq fin_busqueda   @ Si terminamos, salta al final
21
22     ldr r5, [r0]          @ R5 = Lee el valor actual de la memoria
23     cmp r5, r2            @ Compara el valor Actual vs Máximo_Registrado
24     ble siguiente        @ Branch if Less or Equal: Si es menor o igual, ignóralo y salta a 'siguiente'
25
26     @ Si llega a esta linea, encontramos un nuevo mayor
27     mov r3, r5            @ R3 adopta la dirección de memoria de este nuevo mayor
28     mov r2, r5            @ R2 adopta el nuevo valor mayor
29
30 siguiente:
31     add r0, r0, #4         @ Avanza la lectura en la memoria (4 bytes)
32     add r4, r4, #1         @ Incrementamos el contador de ciclo
33     b buscar_mayor        @ Repetimos
34
35 fin_busqueda:
36     ldr r6, =MAX_VAL      @ Carga dirección para guardar el valor
37     str r3, [r6]           @ Almacena en memoria el valor mayor
38     ldr r6, =MAX_DIR      @ Carga dirección para guardar la ubicación
39     str r3, [r6]           @ Almacena en memoria la dirección del mayor
40
41     MOV R7, #1             @ sys_exit
42     SVC 0                 @ Terminar

```

Figura 27: Iteración 8 del ciclo ( $R4 = 0x8$ ). La condición de salto evita que los números menores sobreescriban el valor máximo ya encontrado.

```

20     beq fin_busqueda   @ Si terminamos, salta al final
21
22     ldr r5, [r0]          @ R5 = Lee el valor actual de la memoria
23     cmp r5, r2            @ Compara (Valor_Actual vs Máximo_Registrado)
24     ble siguiente        @ Branch if Less or Equal: Si es menor o igual, ignóralo y salta a 'siguiente'
25
26     @ Si llega a esta linea, encontramos un nuevo mayor
27     mov r3, r5            @ R3 adopta la dirección de memoria de este nuevo mayor
28     mov r2, r5            @ R2 adopta el nuevo valor mayor
29
30 siguiente:
31     add r0, r0, #4         @ Avanza la lectura en la memoria (4 bytes)
32     add r4, r4, #1         @ Incrementamos el contador de ciclo
33     b buscar_mayor        @ Repetimos
34
35 fin_busqueda:
36     ldr r6, =MAX_VAL      @ Carga dirección para guardar el valor
37     str r3, [r6]           @ Almacena en memoria el valor mayor
38     ldr r6, =MAX_DIR      @ Carga dirección para guardar la ubicación
39     str r3, [r6]           @ Almacena en memoria la dirección del mayor
40
41     MOV R7, #1             @ sys_exit
42     SVC 0                 @ Terminar

```

Figura 28: Fin de la ejecución (SVC 0). El valor máximo y su dirección se han almacenado en memoria correctamente.

## Análisis de resultados

Antes de entrar al ciclo de evaluación, el programa establece las condiciones iniciales cargando la dirección base del arreglo en el registro R0, la cual corresponde a 0x40011040. Se carga el



---

límite de iteraciones en R1 con el valor de 0x14 (20 en decimal). Asumiendo que el primer dato es el mayor por defecto, la instrucción LDR R2, [R0] extrae el valor almacenado en esa primera dirección, cargando un 0x5 (5) en el registro R2, mientras que R3 guarda la dirección 0x40011040. El contador R4 se inicializa en 0x1 y el puntero R0 se incrementa en 4 bytes para apuntar al siguiente elemento (0x40011044). Todos estos valores se confirman en los registros de la CPU mostrados en la primera captura.

Al entrar al bucle **buscar\_mayor**, se evalúan secuencialmente los datos mediante la instrucción LDR R5, [R0]. En la segunda iteración, el puntero se encuentra en 0x40011044 y carga el valor 0xC (12) en R5. La instrucción CMP R5, R2 compara este 0xC contra el 0x5 registrado. Como 12 es mayor que 5, no se activa la condición de salto de **BLE siguiente**, permitiendo que el flujo entre al bloque de actualización. En este punto, R2 adopta el nuevo valor máximo (0xC) y R3 adopta su respectiva dirección (0x40011044), tal como se evidencia en la segunda captura donde el PC está detenido justo en la reasignación de direcciones.

A medida que el ciclo avanza, el programa detecta el verdadero valor máximo del arreglo. En la captura de la iteración 6 (R4 = 0x6), se observa que el registro R2 contiene el valor 0x69 (105 en decimal). Este número corresponde al sexto elemento del arreglo original. Consecuentemente, el registro R3 preserva la dirección exacta de este dato, indicando 0x40011054 (calculado como la dirección base 0x40011040 + 5 desplazamientos de 4 bytes). A partir de este punto, en las iteraciones subsecuentes (como se observa en la iteración 8 de la cuarta captura), los valores leídos en R5 resultan ser menores a 0x69. Esto provoca que la instrucción **BLE siguiente** se cumpla de forma continua, saltando la actualización y manteniendo intactos los registros R2 y R3.

El ciclo iterativo finaliza cuando el contador R4 alcanza el valor de 0x14, activando el salto condicional **BEQ fin\_busqueda**. En esta última sección, el programa cumple con los incisos solicitados trasladando los resultados retenidos en el procesador hacia la memoria principal. Se carga la dirección de la variable **MAX\_VAL** en R6 (0x40011090) y mediante STR R2, [R6] se escribe permanentemente el valor 0x69. A continuación, se carga la dirección de **MAX\_DIR** en R6 (0x40011094) y se guarda el contenido de R3 (0x40011054). La última captura valida que el registro R2 retuvo satisfactoriamente el número 105 y el registro R3 su ubicación exacta, demostrando que el manejo de punteros y los saltos condicionales operaron con total precisión sobre la memoria estática.



## Actividad 7

Realizar un programa que ordene de manera ascendente un arreglo de 32 elementos de 32 bits; deberá:

- Mantener el arreglo original.
- Generar otro arreglo con el ordenamiento del original.

**Arreglo original.**

$A[0]$	$A[1]$	$A[2]$	$\cdots$	$A[31]$
--------	--------	--------	----------	---------

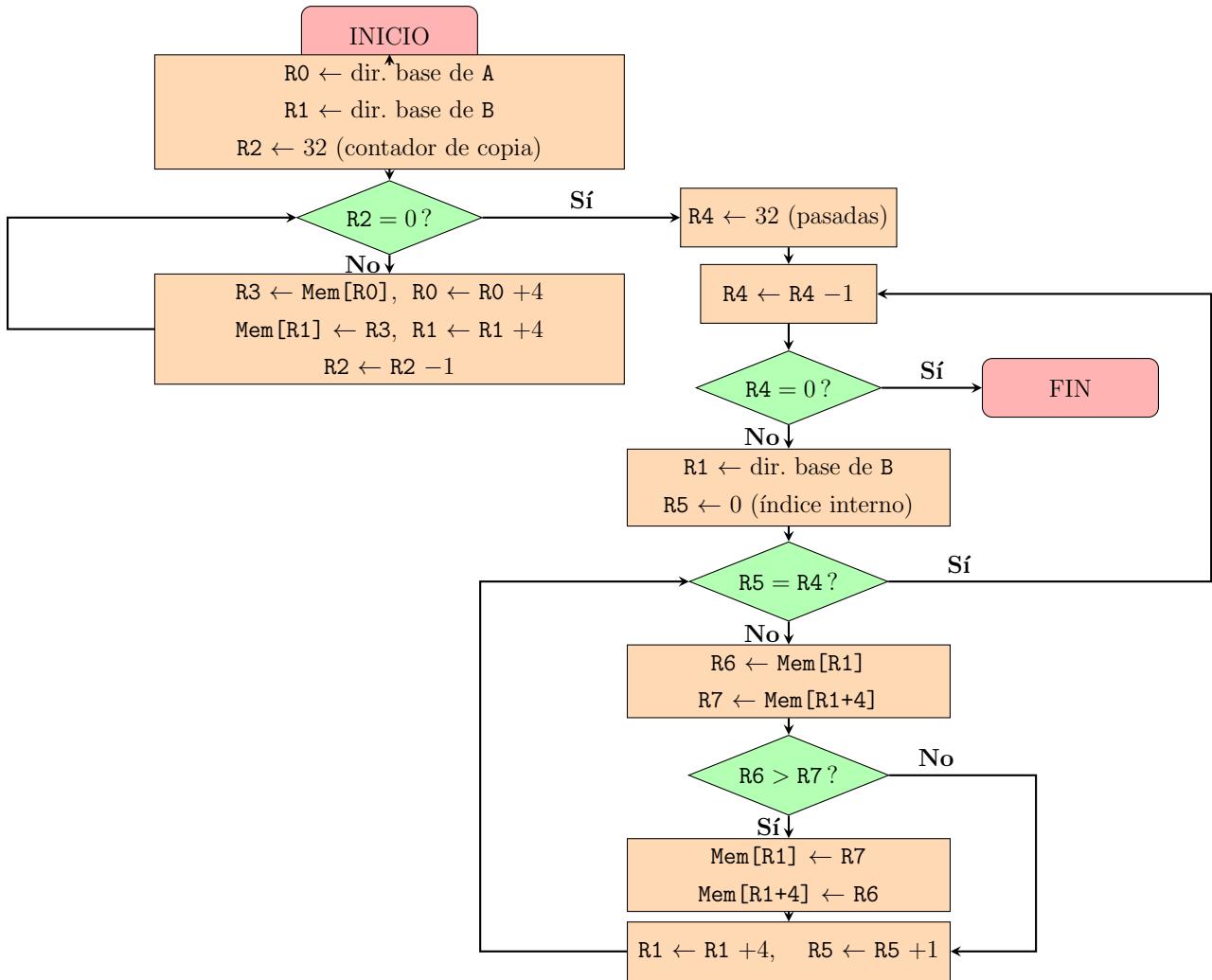
**Arreglo ordenado.**

Menor $A[x]$	Mayor $A[y]$
--------------	--------------

### Propuesta de solución

La solución se estructura en dos fases secuenciales. En la **primera fase** se realiza una copia íntegra del arreglo original A hacia un arreglo auxiliar B, recorriendo los 32 elementos mediante un ciclo con post-indexado que lee de A y escribe en B, avanzando ambos apuntadores en 4 bytes por iteración y decrementando un contador hasta llegar a cero. Una vez completada la copia, la **segunda fase** aplica el algoritmo de ordenamiento burbuja sobre el arreglo B, preservando intacto el arreglo original. El algoritmo utiliza dos bucles anidados: el bucle externo decremente el límite de comparaciones en cada pasada (de 32 hasta 0), mientras que el bucle interno recorre los elementos adyacentes de B comparándolos entre sí; cuando el elemento izquierdo es mayor que el derecho, se efectúa un intercambio cruzado mediante instrucciones STR que escriben los valores en posiciones invertidas. Este proceso se repite hasta que el bucle externo agota sus pasadas, dejando el arreglo B completamente ordenado de menor a mayor.

A continuación se presenta el diagrama de flujo correspondiente al algoritmo descrito:



El diagrama se divide en dos fases claramente diferenciadas. En la **Fase 1** se inicializan los apuntadores  $R0$  (origen en A) y  $R1$  (destino en B) junto con el contador  $R2$  con valor 32. El ciclo de copia evalúa si  $R2$  ha llegado a cero; mientras no lo sea, se lee un dato de A mediante post-indexado (incrementando  $R0$ ), se escribe en B (incrementando  $R1$ ), y se decrementa el contador, repitiendo el proceso. Cuando  $R2$  alcanza cero, los 32 elementos han sido transferidos y el flujo pasa a la Fase 2.

En la **Fase 2** se inicializa  $R4$  con 32 para controlar las pasadas del algoritmo burbuja. Al inicio de cada pasada se decrementa  $R4$ ; si llega a cero, el ordenamiento está completo y el programa termina. En caso contrario, se reinicializa el apuntador  $R1$  a la base de B y el índice interno  $R5$  a cero. El bucle interno compara  $R5$  con  $R4$ : si son iguales, la pasada terminó y



se regresa al bucle externo para decrementar R4 nuevamente. De lo contrario, se cargan los dos elementos adyacentes B[i] y B[i+1] en R6 y R7 respectivamente. Si R6 es mayor que R7, se ejecuta el intercambio cruzado escribiendo los valores en posiciones invertidas; si no, se omite el intercambio. Finalmente, se avanza el apuntador R1 en 4 bytes y se incrementa R5, repitiendo el bucle interno hasta completar la pasada.

## Desarrollo

Listing 7: Código de la Actividad 7

```
1  /* ACTIVIDAD 7: Ordenamiento Burbuja de 32 elementos (32 bits)
2   Objetivo: Conservar arreglo original, ordenar la copia.
3 */
4 .data
5   @ Arreglo original desordenado (32 elementos)
6   A: .word
7     32,31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,
8     12,11,10,9,8,7,6,5,4,3,2,1
9   @ Arreglo copia donde se hará el ordenamiento
10  B: .skip 128           @ Reserva 128 bytes (32 words x 4)
11
12 .text
13 .global main
14
15 main:
16   @ --- FASE 1: COPIAR A en B ---
17   ldr r0, =A             @ R0 apunta a Original
18   ldr r1, =B             @ R1 apunta a Copia
19   mov r2, #32            @ R2 contador para copiar
20
21 copiar:
22   cmp r2, #0             @ ¿Quedan elementos por copiar?
23   beq iniciar_orden      @ Si es 0, terminamos de copiar y
24   vamos a ordenar
25   ldr r3, [r0], #4        @ Lee de A y avanza
26   str r3, [r1], #4        @ Escribe en B y avanza
27   sub r2, r2, #1          @ Resta 1 al contador
28   b copiar
```

```

26
27     @ --- FASE 2: ORDENAMIENTO BURBUJA (Sobre B) ---
28
29     iniciar_orden:
30         mov r4, #32                      @ R4 = N (Cantidad total de elementos)
31
32     bucle_externo:
33         subs r4, r4, #1                  @ Resta 1 a N (N = N - 1) y actualiza
34             flags (S final)
35         beq fin_ordenamiento          @ Si N llega a 0, todo está ordenado
36
37         ldr r1, =B                     @ Resetea el puntero R1 al inicio de B
38             para cada pasada
39         mov r5, #0                      @ R5 = 'i' (Índice del bucle interno)
40
41     bucle_interno:
42         cmp r5, r4                  @ Compara el índice interno 'i' con 'N
43             ,
44         beq bucle_externo           @ Si i == N, terminó esta pasada,
45             regresa al bucle externo
46
47         ldr r6, [r1]                 @ R6 = B[i] (Valor actual)
48         ldr r7, [r1, #4]              @ R7 = B[i+1] (Valor adyacente derecho
49             )
50
51         cmp r6, r7                  @ Comparamos si el actual es mayor que
52             el derecho
53         ble no_cambiar            @ Branch if Less or Equal: Si B[i] <=
54             B[i+1] están bien, no cambies
55
56         @ Si llegamos aquí, B[i] es mayor, tenemos que hacer INTERCAMBIO
57             (Swap)
58         str r7, [r1]                 @ Escribimos el valor menor (R7) en la
59             posición izquierda B[i]
60         str r6, [r1, #4]              @ Escribimos el valor mayor (R6) en la
61             posición derecha B[i+1]
62
63     no_cambiar:

```

```

52      add r1, r1, #4          @ Avanzamos el puntero de memoria para
53      evaluar los siguientes
54      add r5, r5, #1          @ i++
55      b bucle_interno        @ Repetimos el bucle interno
56
57 fin_ordenamiento:
58     MOV R7, #1              @ sys_exit
59     SVC 0                  @ Fin

```

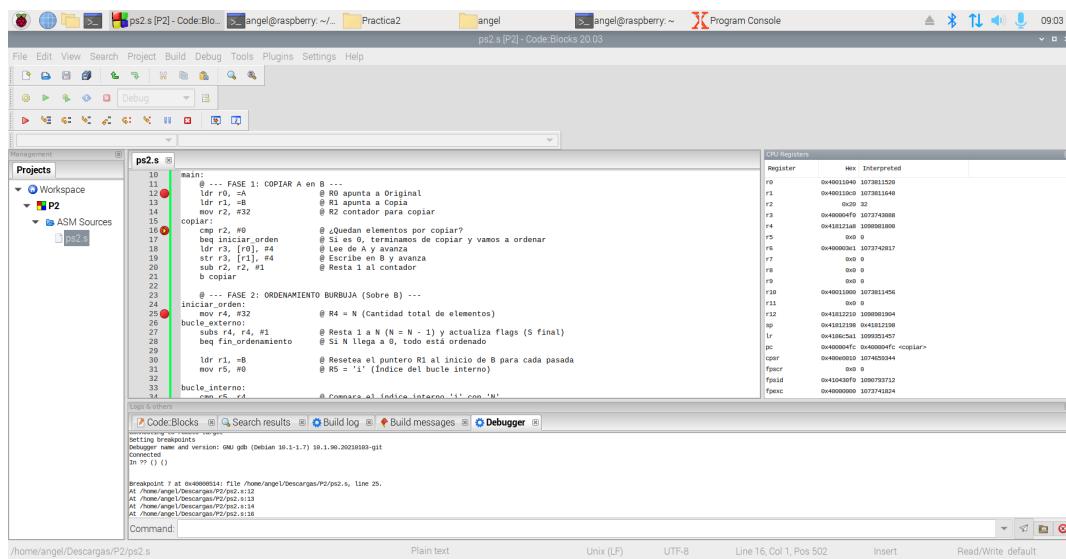


Figura 29: Fase 1 (Copiado). Inicialización de los apuntadores base para el arreglo original (R0) y el arreglo copia (R1).



Figura 30: Primera iteración del ciclo de copiado. El registro R3 extrae el primer valor a transferir.

Figura 31: Avance del ciclo de copiado en la iteración correspondiente al quinto elemento transferido ( $R2 = 0x1C$ ).



The screenshot shows the Code::Blocks IDE interface. The left pane displays the assembly code for file ps2.s, which contains two main phases: copying from A to B and bubble sort. The right pane shows the CPU Registers window with various registers and their values. The bottom status bar indicates the file is ps2.s [P2] - Code Blo., the project is angel, and the time is 09:07.

```
ps2.s
10 main: ... FASE 1: COPIAR A en B ...
11     ldr r0, =A          @ R0 apunta a Original
12     ldr r1, =B          @ R1 apunta a Copia
13     mov r2, #32         @ R2 contador para copiar
14
15     copiar:             @ ...
16         cmp r2, #0        @ Quedan elementos por copiar?
17         bne fin_orden    @ Si es cero terminamos de copiar y vamos a ordenar
18         ldr r3, [r0], #4    @ Lee de A y avanza
19         str r3, [r1], #4    @ Escribe en B y avanza
20         add r0, #1          @ Resta 1 al contador
21         b copiar           @ ...
22
23     @ ... FASE 2: ORDENAMIENTO BURBUJA (Sobre B) ...
24     inicial_orden:       @ ...
25         mov r4, #32         @ R4 = N (Cantidad total de elementos)
26         buble_externo:      @ ...
27         subs r4, r4, #1      @ Resta 1 a N (N - 1) y actualiza flags (S final)
28         beq fin_ordenamiento @ Si N llega a 0, todo està ordenado
29
30         ldr r1, =B          @ Resetas el puntero R1 al inicio de B para cada pasada
31         mov r5, #0
32
33         buble_interno:      @ ...
34         cmp r5, r4          @ Compara el indice interno 'i' con 'N'
35         bne buble_externo   @ Si i == N, terminó esta pasada, regresa al bucle externo
36
37         ldr r0, =A            @ R0 = A[1].Valor_actuall
```

Register	Hex	Interpreted
r0	0x40011000	1073811648
r1	0x40011100	1073811776
r2	0x0	0
r3	0x1	1
r4	0x41011000	1073811800
r5	0x0	0
r6	0x40000000	1073742817
r7	0x0	0
r8	0x0	0
r9	0x0	0
r10	0x40011000	1073811456
r11	0x41011000	1073811204
r12	0x41011000	1073811204
sp	0x415e2100	1073812108
lr	0x415e0101	1073811407
pc	0x40000514	1073810514 <iniciar_orden>
spcr	0x0	0
fpcr	0x0	0
fpcid	0x41000000	1073813712
fpxc	0x40000000	1073741624

Figura 32: Fin de la fase 1 e inicio de la fase 2. Los apuntadores han recorrido 128 bytes en memoria.

The screenshot shows the Code::Blocks IDE interface. The left pane displays the assembly code for file ps2.s, which contains two main phases: copying from A to B and bubble sort. The right pane shows the CPU Registers window with various registers and their values. The bottom status bar indicates the file is ps2.s [P2] - Code Blo., the project is angel, and the time is 09:08.

```
ps2.s
13     ldr r1, =B          @ R1 apunta a Copia
14     mov r2, #32         @ R2 contador para copiar
15
16     copiar:             @ ...
17         cmp r2, #0        @ Quedan elementos por copiar?
18         bne fin_orden    @ Si es cero terminamos de copiar y vamos a ordenar
19         ldr r3, [r0], #4    @ Lee de A y avanza
20         str r3, [r1], #4    @ Escribe en B y avanza
21         sub r2, r2, #1      @ Resta 1 al contador
22
23     @ ... FASE 2: ORDENAMIENTO BURBUJA (Sobre B) ...
24     inicial_orden:       @ ...
25         mov r4, #32         @ R4 = N (Cantidad total de elementos)
26         buble_externo:      @ ...
27         sub r4, r4, #1      @ Resta 1 a N (N - 1) y actualiza flags (S final)
28         beq fin_ordenamiento @ Si N llega a 0, todo està ordenado
29
30         ldr r1, =B          @ Resetas el puntero R1 al inicio de B para cada pasada
31         mov r5, #0
32
33         buble_interno:      @ ...
34         cmp r5, r4          @ Compara el indice interno 'i' con 'N'
35         bne buble_externo   @ Si i == N, terminó esta pasada, regresa al bucle externo
36
37         ldr r0, =A            @ R0 = A[1].Valor_actuall
```

Register	Hex	Interpreted
r0	0x40011000	1073811648
r1	0x40011100	1073811776
r2	0x0	0
r3	0x1	1
r4	0x41011000	1073811456
r5	0x0	0
r6	0x0	0
r7	0x0	0
r8	0x0	0
r9	0x0	0
r10	0x40000000	1073742817
r11	0x0	0
r12	0x41011000	1073811204
sp	0x415e2100	1073812108
lr	0x415e0101	1073811407
pc	0x40000514	1073810514 <iniciar_orden>
spcr	0x0	0
fpcr	0x0	0
fpcid	0x41000000	1073813712
fpxc	0x40000000	1073741624

Figura 33: Inicialización del bucle externo del algoritmo de burbuja. R1 se reinicia a la dirección base de la copia.



The screenshot shows the Code::Blocks IDE interface with the following details:

- File Menu:** File, Edit, View, Search, Project, Build, Debug, Tools, Plugins, Settings, Help.
- Project Manager:** Projects, Workspace, P2, ASM Sources, ps2.s.
- Code Editor:** The main editor displays assembly code for file `ps2.s`. The code includes comments explaining the logic of the program, such as resetting pointers, comparing indices, and handling loops. It also includes assembly instructions like `ldr`, `cmp`, and `str`.
- Registers Window:** Shows the CPU registers (R0-R15, PC, SP) and their hex/interpreted values. For example, R0 is 0x40011000 and R1 is 0x40011004.
- Code Block Status Bar:** Shows tabs for Code, Results, Search results, Build log, Build messages, and Debugger.
- Command Line:** At the bottom, the terminal window shows the command `arm-none-eabi-objdump -D ps2.s` being run.

Figura 34: Carga de elementos adyacentes en R6 y R7 para su respectiva comparación.

Figura 35: Ejecución del intercambio (*swap*) al detectar que el elemento izquierdo es mayor que el derecho.



The screenshot shows the Code::Blocks IDE interface with the following details:

- Title Bar:** ps2.s [P2] - Code.Blo... > angel@raspberry:~/... Practica2 angel > angel@raspberry:~ X Program Console
- Menu Bar:** File Edit View Search Project Build Debug Tools Plugins Settings Help
- Toolbar:** Includes icons for New, Open, Save, Print, Run, Stop, and Break.
- Project Explorer:** Shows a workspace named "P2" containing "ASM Sources" and "ps2.s".
- Code Editor:** Displays assembly code for a bubble sort algorithm. The code includes comments explaining the logic of comparing adjacent elements and swapping them if they are in the wrong order. It also handles boundary conditions and loops.
- Registers Window:** Shows the CPU registers (r9 to r15) and their corresponding memory addresses and values.
- Stack Window:** Shows the stack contents, including the current PC value.
- Output Window:** Displays assembly language mnemonics and their corresponding opcodes.

Figura 36: Actualización del apuntador interno R1 y del contador interno R5 antes de la siguiente comparación.

The screenshot shows the Code::Blocks IDE interface with multiple windows open. The main window displays assembly code for a swap operation between memory locations B[1] and B[2]. The assembly code includes comments in Spanish explaining the logic: comparing indices, handling boundary conditions, and performing the swap. The assembly code is as follows:

```
ps2.s
32    bucle_interno;
33    cmp r5, r4          ; Compara el indice interno 'i' con 'N'
34    beq bucle_externo   ; Si i == N, terminó esta pasada, regresa al bucle externo
35
36    ldr r7, [r1]          ; R7 = B[i] (Valor actual)
37    ldr r7, [r1, #4]      ; R7 = B[i+1] (Valor adyacente derecho)
38
39    cmp r6, r7          ; Comparamos si el actual es mayor que el derecho
40    bne no_cambiar       ; Branch if Less or Equal: SI [i] <= B[i+1] estan bien, no cambies
41
42    ; Si llegamos aqui, B[i] es mayor, tenemos que hacer INTERCAMBIO (Swap)
43    str r7, [r1]          ; Escribimos el valor menor (R7) en la posicion izquierda B[i]
44    str r6, [r1, #4]      ; Escribimos el valor mayor (R6) en la posicion derecha B[i+1]
45
46    no_cambiar:
47    add r1, r1, #4        ; Avanzamos el puntero de memoria para evaluar los siguientes
48    add r5, r5, #4
49    b bucle_interno
50
51 fin_operacion:
52    MOV R7, #1
53    SVC 0
54
```

The Registers window shows the state of the CPU registers:

Register	Hex	Interpreted
r0	0x40011000	1073011440
r1	0x40011004	1073011000
r2	0x0	0
r3	0x1	1
r4	0x0	0
r5	0x1	1
r6	0x2	2
r7	0x1	1
r8	0x0	0
r9	0x0	0
r10	0x40012000	1073011456
r11	0x0	0
r12	0x40182210	1098901904
sp	0x40182210	1098901904
pc	0x40000000	<fn_operacion>+0
cper	0x00000010	1611530256
fpcsr	0x0	0
fpsid	0x41003f00	1007073712
fpcexc	0x00000000	1073741824

The Log window shows the following output:

```
Code::Blocks  Search results  Build log  Build messages  Debugger
```

```
Setting SHELL to "/bin/sh"
Setting TERM to "xterm"
Connecting to remote target
Detected target: arm-none-eabi
Debugger name and version: GDB (Debian 10.1-1.7) 10.1-90.20201005-git
Version: 10.1-90.20201005-git
In ???
At /home/angel/Desktop/P2/ps2.s:12
Breakpoint 1 set at pc 0x4000004f.
Temporary breakpoint 2 at pc 0x4000004f.
Temporary breakpoint 3 at pc 0x4000004f.
Temporary breakpoint 4 at pc 0x4000004f.
Temporary breakpoint 5 also set at pc 0x4000004f.
Temporary breakpoint 6 at pc 0x4000004f.
Temporary breakpoint 7 at pc 0x4000004f.
Temporary breakpoint 8 at pc 0x4000004f.
Temporary breakpoint 9 at pc 0x4000004f.
Temporary breakpoint 10 at pc 0x4000004f.
Temporary breakpoint 11 at pc 0x4000004f.
Temporary breakpoint 12 at pc 0x4000004f.
Temporary breakpoint 13 at pc 0x4000004f.
Temporary breakpoint 14 at pc 0x4000004f.
Temporary breakpoint 15 at pc 0x4000004f.
Temporary breakpoint 16 at pc 0x4000004f.
Temporary breakpoint 17 at pc 0x4000004f.
Temporary breakpoint 18 at pc 0x4000004f.
Temporary breakpoint 19 at pc 0x4000004f.
Temporary breakpoint 20 at pc 0x4000004f.
Temporary breakpoint 21 at pc 0x4000004f.
Temporary breakpoint 22 at pc 0x4000004f.
Temporary breakpoint 23 at pc 0x4000004f.
Temporary breakpoint 24 at pc 0x4000004f.
Temporary breakpoint 25 at pc 0x4000004f.
Temporary breakpoint 26 at pc 0x4000004f.
Temporary breakpoint 27 at pc 0x4000004f.
Temporary breakpoint 28 at pc 0x4000004f.
Temporary breakpoint 29 at pc 0x4000004f.
Temporary breakpoint 30 at pc 0x4000004f.
Temporary breakpoint 31 at pc 0x4000004f.
Temporary breakpoint 32 at pc 0x4000004f.
Temporary breakpoint 33 at pc 0x4000004f.
Temporary breakpoint 34 at pc 0x4000004f.
Temporary breakpoint 35 at pc 0x4000004f.
Temporary breakpoint 36 at pc 0x4000004f.
```

The Command window shows the command entered: `./ps2.s`.

Figura 37: Finalización del programa. Los contadores de ambos bucles agotaron sus iteraciones.

## Análisis de resultados

La ejecución inicia configurando la transferencia de datos. Las instrucciones LDR R0, =A y LDR R1, =B asignan las direcciones base de ambos arreglos. Como se observa en la primera



captura, R0 almacena la dirección 0x40011040 (inicio de A) y R1 almacena 0x400110C0 (inicio de B). Se establece el límite de 32 elementos en R2 (0x20). Al entrar a la etiqueta **copiar**, la instrucción LDR R3, [R0], #4 carga el primer elemento del arreglo (0x20 o 32 en decimal) en R3 y avanza el apuntador original a 0x40011044. Seguidamente, STR R3, [R1], #4 guarda este valor en la dirección apuntada por R1 y desplaza dicho apuntador a 0x400110C4. Este avance coordinado de ambos apuntadores se mantiene constante, comprobándose en las capturas 2 y 3, hasta que el contador R2 llega a cero, momento en el que R1 alcanza la dirección 0x40011140 (exactamente 128 bytes de desplazamiento, calculados como  $32 \times 4$ ).

Una vez completada la copia intacta, inicia la fase de ordenamiento bajo la etiqueta **iniciar\_orden**. Se carga la constante de 32 elementos en R4 para fungir como el límite decreciente del bucle externo. La instrucción SUBS R4, R4, #1 resta una unidad en cada pasada general y actualiza la bandera de estado para verificar si se alcanzó el límite. En la captura 5, se observa que R4 adquiere el valor de 0x1F (31). Es vital notar la instrucción LDR R1, =B, la cual re-inicializa el apuntador R1 a la dirección base 0x400110C0 al inicio de cada pasada, mientras que R5 se limpia con 0x0 para funcionar como el índice del bucle interno.

Dentro del **bucle\_interno**, las instrucciones LDR R6, [R1] y LDR R7, [R1, #4] acceden simultáneamente a dos valores adyacentes en la memoria de la copia. En la primera evaluación (captura 6), R6 carga el valor 0x20 (32) correspondiente a B[0], y R7 carga 0x1F (31) correspondiente a B[1]. La instrucción CMP R6, R7 compara ambos registros. Puesto que 32 es mayor que 31, no se cumple la condición BLE **no\_cambiar**, por lo que el programa prosigue a intercambiar los datos en memoria.

El intercambio se efectúa de manera cruzada: la instrucción STR R7, [R1] toma el valor menor (31) y lo sobreescribe en la dirección inferior, mientras que STR R6, [R1, #4] toma el valor mayor (32) y lo aloja en la dirección superior inmediata. Esto se observa en progreso en la captura 7. Después del intercambio, bajo la etiqueta **no\_cambiar**, se incrementa el apuntador base con ADD R1, R1, #4 (desplazando a 0x400110C4 como se ve en la captura 8) y se aumenta el índice interno R5 en una unidad. Este patrón de comparación y desplazamiento en la memoria se repite iterativamente hasta que el bucle interno alcanza al bucle externo, empujando los valores más altos hacia las direcciones de memoria más elevadas, lo que a nivel del procesador cumple satisfactoriamente el algoritmo de burbuja y satisface todas las directrices establecidas.



## 2. Conclusiones:

- **Espinoza Matamoros Percival Ulises:** Gracias a la realización de la práctica así como de las actividades planteadas a desarrollar a lo largo de la misma, pude conocer y aplicar las diferentes variantes del modo de direccionamiento indirecto que existen en los procesadores ARM, siendo este conocimiento fundamental para la programación de procesadores ARM ya que emplear estas instrucciones permite tener un código más eficiente y legible cuando se trabaja con arreglos, siendo una estructura de datos fundamental a la hora de implementar cualquier solución por medio de la programación. El hacer uso de las variantes pre-indexadas y post-indexadas permite realizar dos operaciones diferentes, acceder a la memoria y actualizar el puntero, por lo que emplear dichas instrucciones ayuda a reducir el tamaño del código del CPU lo cual se puede ver reflejado en el rendimiento del programa. Adicionalmente se tiene otra variante la cual es el offset simple la cual aunque no actualiza el registro base, son de gran utilidad para desplazarse en bloques continuos de memoria de forma fácil.
- **Flores Colin Victor Jaziel:** En esta segunda práctica de laboratorio mediante la realización de las actividades propuestas por el manual me permitió validar la eficiencia de la arquitectura ARM al manipular estructuras de datos complejas mediante el uso de los modos de direccionamiento indirecto, pre-indexado y post-indexado por medio de la implementación y experimentación de las instrucciones LDR y STR.

El uso del modo post-indexado (usado en la Actividad 2 y 3) demostró su versatilidad en el recorrido de arreglos ya que integrar la actualización del puntero (auto-incremento de  $\pm 4$  bytes) se pudo reducir el número de instrucciones.

El uso de la instrucción LSL me facilitó el acceso aleatorio y el cálculo de direcciones en estructuras bidimensionales como matrices y los apuntadores indirectos optimizaron la búsqueda y ordenamiento de datos al permitir comparaciones fluidas entre elementos en las últimas 2 actividades.

- **Lara Hernandez Angel Husiel:**



---

## Referencias

- Anaya, R. (s.f.). *Manual de recursos y aplicaciones Plataforma Raspberry Pi*. <https://odin.filb.unam.mx/micros/docs/Tutoriales%20Raspberry.pdf>
- Elahi, A. (2022, 17 de marzo). *Computer systems: Digital Design, Fundamentals of Computer Architecture and ARM Assembly Language* (2.<sup>a</sup> ed.). Springer. <https://doi.org/10.1007/978-3-030-93449-1>
- Harris, D., & Harris, S. (2015, 22 de abril). *Digital Design and Computer Architecture* (Arm Edition). Morgan Kaufmann Pub.
- Smith, S. (2019, octubre). *Raspberry Pi Assembly Language Programming*. Apress. <https://doi.org/10.1007/978-1-4842-5287-1>