



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.

Propuesta de solución

Desarrollo

Listing 1: Código de la Actividad 1

```
1  /* ACTIVIDAD 1: Direccionamiento 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
9 .text
10 .global main          @ Define 'main' como global para Code
11   ::Blocks / Linker
12
13 main:
14   ldr r1, =i            @ Carga en R1 la dirección base de la
15   variable 'i'
16   mov r2, #0             @ R2 será nuestro contador,
17   inicializado en 0
18
19 loop:
20   cmp r2, #16           @ Compara el contador R2 con el límite
21   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
```

Análisis de resultados



Actividad 2

Modificar el programa de la actividad 1, para usar el direccionamiento indexado de su preferencia con el doble de datos.

Propuesta de solución

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



23	MOV R7 , #1	<i>@ Prepara sys_exit</i>
24	SVC 0	<i>@ Termina ejecución</i>

Análisis de resultados



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.

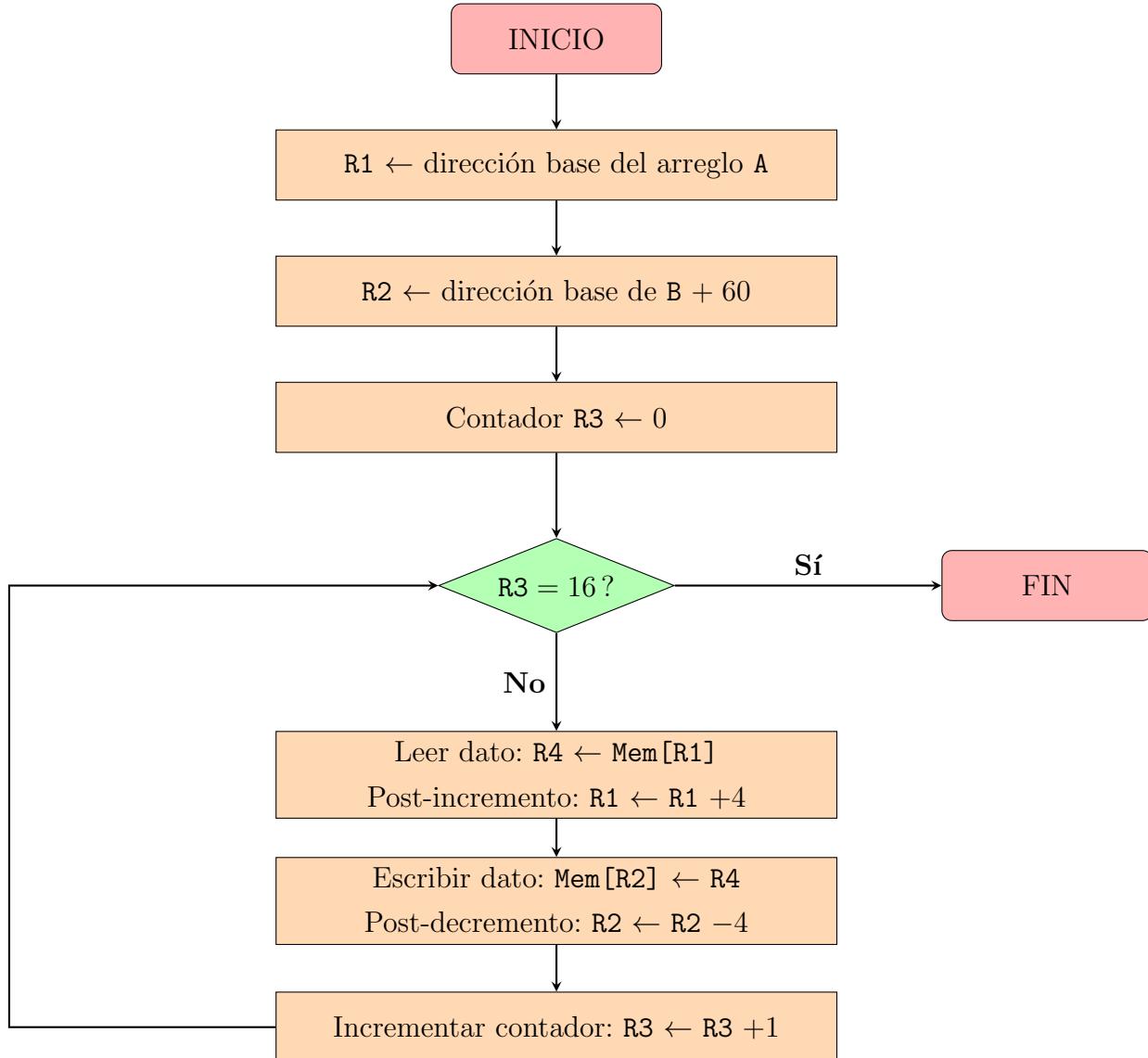
$$A = [\text{dato}_1, \text{dato}_2, \text{dato}_3, \text{dato}_4, \dots, \text{dato}_{15}, \text{dato}_{16}] \quad @\text{Original}$$

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

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
```

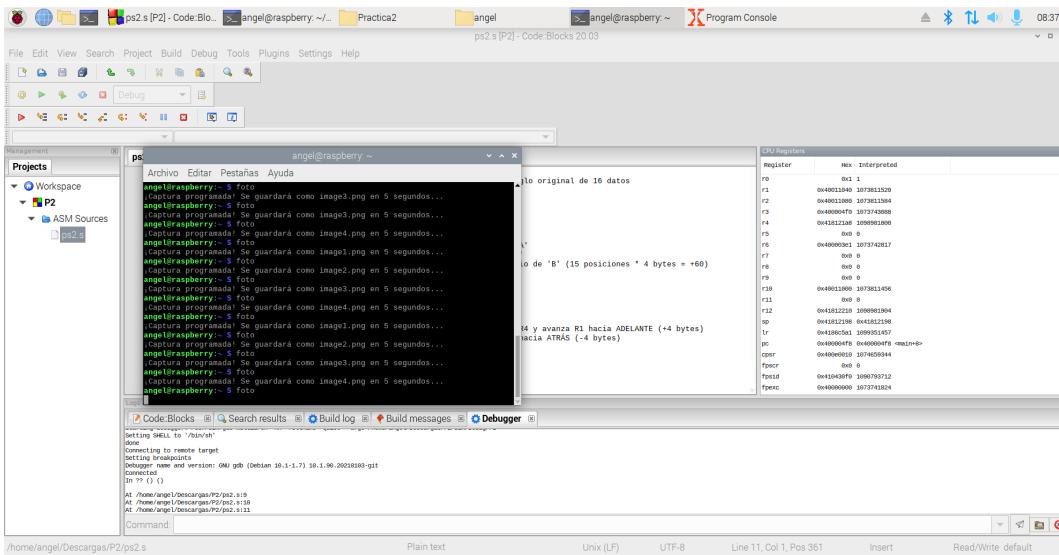


Figura 1: Entorno Code::Blocks preparado para iniciar la depuración del código fuente.

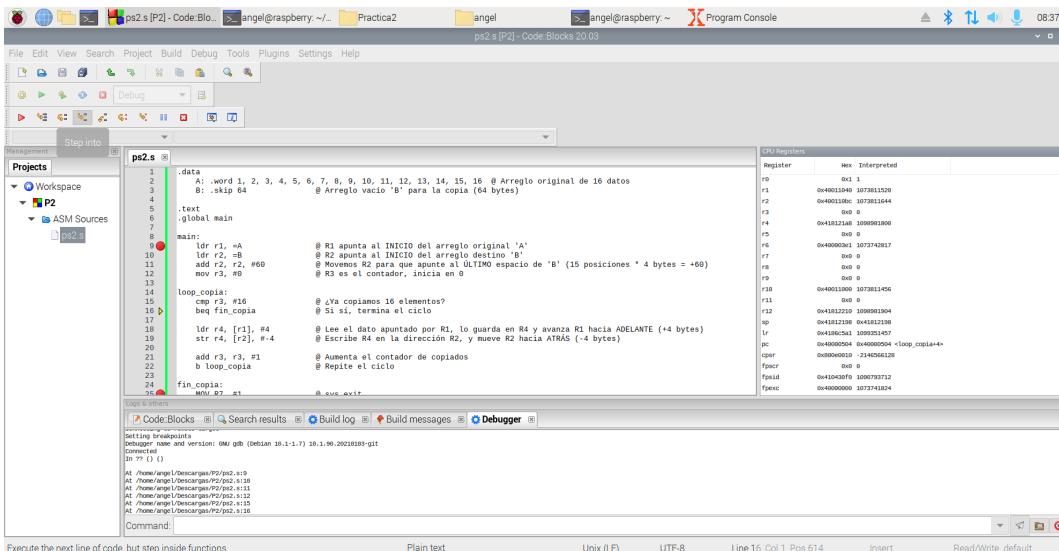


Figura 2: Inicialización de apuntadores. Se observa que R1 apunta al inicio de A (0x40011040) y R2 apunta al final de B (0x400110bc). El contador R3 inicia en 0.

```

ps2.s [P2] - Code Blo... angel@raspberry:~/Practica2 angel ps2.s [P2] - Code-Blocks 20.03
File Edit View Search Project Build Debug Tools Plugins Settings Help
File Edit View Search Project Build Debug Tools Plugins Settings Help
Projects P2
ASM Sources 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 ÚLTIMO espacio de 'B' (15 posiciones * 4 bytes = +60)
12    mov r3, #0 @ R3 es el contador, incide en 0
13
14    loop_copia:
15        cmp r3, #16 @ Ya copiamos 16 elementos?
16        beq fin_copia @ Si 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 muerde R2 hacia ATRÁS (-4 bytes)
20
21        add r3, r3, #1 @ Aumenta el contador de copiados
22        b loop_copia @ Repite el ciclo
23
24    fin_copia:
25        mnw r2, #1
26    .euc_exit:

```

Code-Blocks | Search results | Build log | Build messages | Debugger |

/home/angel/Desktop/P2/ps2.s

Plain text Unix (LF) UTF-8 Line 18, Col 1, Pos 672 Insert Read/Write default

Figura 3: Evaluación de la condición de salida (CMP r3, #16) en las primeras fases del ciclo. La ejecución entra al bloque de copiado.

```

ps2.s [P2] - Code Blo... angel@raspberry:~/Practica2 angel ps2.s [P2] - Code-Blocks 20.03
File Edit View Search Project Build Debug Tools Plugins Settings Help
File Edit View Search Project Build Debug Tools Plugins Settings Help
Projects P2
ASM Sources 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 ÚLTIMO espacio de 'B' (15 posiciones * 4 bytes = +60)
12    mov r3, #0 @ R3 es el contador, incide en 0
13
14    loop_copia:
15        cmp r3, #16 @ Ya copiamos 16 elementos?
16        beq fin_copia @ Si 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 muerde R2 hacia ATRÁS (-4 bytes)
20
21        add r3, r3, #1 @ Aumenta el contador de copiados
22        b loop_copia @ Repite el ciclo
23
24    fin_copia:
25        mnw r2, #1
26    .euc_exit:

```

Code-Blocks | Search results | Build log | Build messages | Debugger |

Execute the next line of code Plain text Unix (LF) UTF-8 Line 21, Col 1, Pos 887 Insert Read/Write default

Figura 4: 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.



The screenshot shows the ps2s assembly editor interface. The main window displays the assembly code for the file `ps2.s`. The code defines a global variable `A` (16 bytes), initializes it with values 1 through 16, and then copies it into another variable `B` (also 16 bytes). The assembly code uses R1, R2, R3, and R4 registers, along with memory locations `[R1]`, `[R2]`, and `[R3]`. The editor also shows the CPU Registers pane, which lists the state of all general-purpose registers (R0-R15) and floating-point registers (F0-F15). The bottom status bar indicates the current file is `ps2.s` and the build status is "Build successful".

```
ps2.s .data A: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 @ Arreglo original de 16 datos B: .skip 64 @ Arreglo vacío "B" para la copia (64 bytes) .text .global main main: ldr r1, =A @ R1 apunta al INICIO del arreglo original 'A' ldr r2, =B @ R2 apunta al INICIO del arreglo destino 'B' add r2, r2, #60 @ Movemos R2 para que apunte al espacio de 'B' (15 posiciones * 4 bytes = +60) mov r3, #0 @ R3 es el contador, inicia en 0 long_copia: cmp r3, #16 @ ¿Ya copiamos 16 elementos? beq fin_copia @ Si sí, termina el ciclo ldr r4, [r1], #4 @ Lee el dato apuntado por R1, lo guarda en R4 y avanza R1 hacia ADELANTE (+4 bytes) str r4, [r2], #4 @ Escribe R4 en la dirección R2, y mueve R2 hacia ATRAS (-4 Bytes) add r3, r3, #1 @ Aumenta el contador de copiados b long_copia @ Repite el ciclo fin_copia: WHTD .exit .LFB0 .end
```

Register	Hex	Interpreted
R0	0x1	0x1
R1	0x40000001 1073811548	0x40000001 1073811548
R2	0x40000001 1073811548	0x40000001 1073811548
R3	0x0	0x0
R4	0x0	0x0
R5	0x0	0x0
R6	0x40000001 1073812017	0x40000001 1073812017
R7	0x0	0x0
R8	0x0	0x0
R9	0x0	0x0
R10	0x40000000 1073811456	0x40000000 1073811456
R11	0x0	0x0
R12	0x40000000 1073811548	0x40000000 1073811548
lr	0x41023100 1099321547	0x41023100 1099321547
pc	0x40000014 1099300514	0x40000014 1099300514
cper	0x00000000 -2140646128	0x00000000 -2140646128
fper	0x0	0x0
result	0x40000000 1073741824	0x40000000 1073741824
fpcr	0x40000000 1073741824	0x40000000 1073741824

Figura 5: 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.

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 title bar indicates the project is 'ps2.s [P2] - Code Blo' and the file is 'angel@raspberry:~/.../Practica2'. The main window displays the assembly code for 'ps2.s' (P2.s) under the 'ASM Sources' tab. The code copies elements from array 'A' to array 'B' and increments a counter. The assembly code is annotated with comments in Spanish. Below the code editor is the 'Log & Editors' panel, which contains tabs for Code Block, Search results, Build log, Build messages, and Debugger. The 'Debugger' tab is selected. The bottom status bar shows the current file path as '/home/angel/Desktop/practicas/P2/ps2.s' and the line number '1:16'. The bottom right corner shows the terminal window with the title 'Program Console'.

Figura 6: 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



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**).

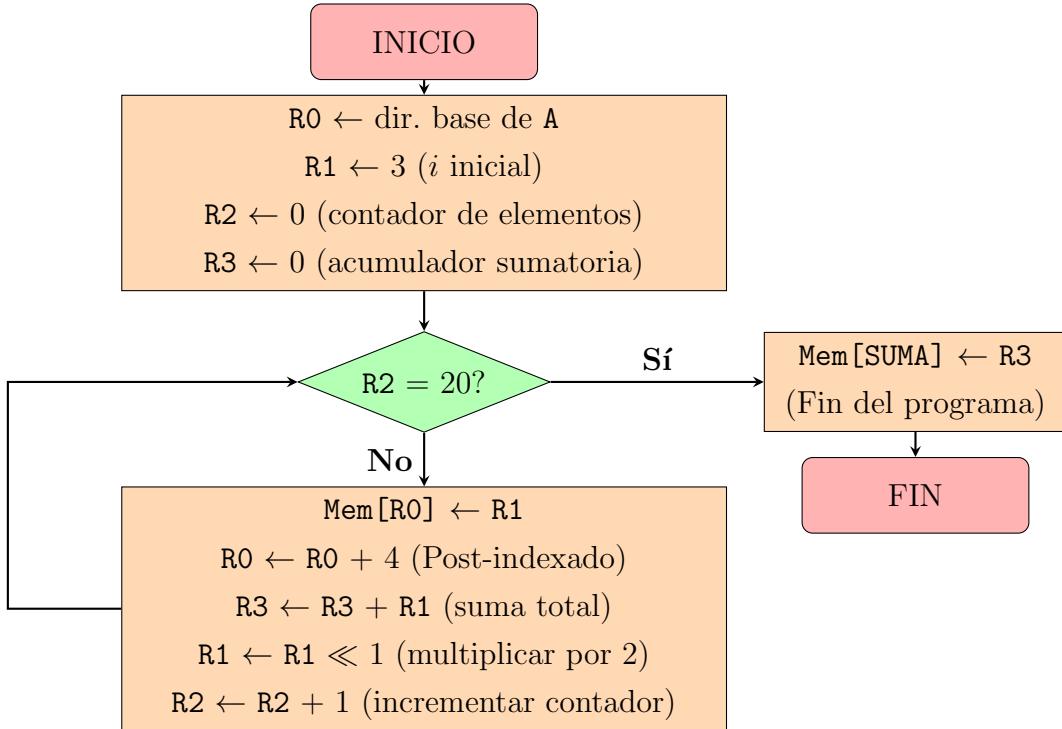
Una vez reservado el espacio es necesario cargar las direcciones de memoria para el correcto funcionamiento del programa, de forma que es necesario cargar la dirección de memoria donde empieza el arreglo A, esta dirección de memoria se guardará en el registro R0, posteriormente en el registro R1 se indicara 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. On the left, the project tree displays a workspace named 'P2' containing an assembly source file 'ps2.s'. The assembly code is shown in the main editor window, with comments explaining the purpose of each instruction. To the right, the 'CPU Registers' window lists the state of all general-purpose registers (r0 to r15), along with stack pointer (sp), base register (lr), program counter (pc), and floating-point registers (fpscr, fpuid, fpexc). Below the registers, the 'Logs & others' window contains a terminal session showing the connection to a remote target (gdb) and the assembly code being run. The bottom status bar indicates the current file is 'ps2.s', the encoding is 'Plain text', and the cursor is at line 12, column 422.

Figura 7: Estado de los registros. Dirección de memoria inicial del arreglo A



The screenshot shows the Code::Blocks IDE interface. The main window displays assembly code for the file ps2.s. The code includes comments explaining the purpose of each instruction, such as initializing registers R0, R1, R2, and R3, and setting up a loop to calculate powers of 2. The CPU Registers window on the right shows the initial state of the registers. The assembly code starts with a global label main and proceeds through various loops and calculations.

Register	Hex	Interpreted
r0	0x40011044	1073741524
r1	0x3	3
r2	0x0	0
r3	0x0	0
r4	0x41812108	1099861860
r5	0x0	0
r6	0x40000003	1073742817
r7	0x0	0
r8	0x0	0
r9	0x0	0
r10	0x40011000	1073741520
r11	0x0	0
r12	0x41812100	1099861864
sp	0x41812100	1099861864
lr	0x4186c541	1099351457
pc	0x40000000	1073741520 <loop_potencias+12>
cpsr	0x00000010	-214056128
fpscr	0x0	0
fpsid	0x410430f0	1096793712
fpexc	0x40000000	1073741824

Figura 8: Estado de los registros. Inicio de la etiqueta `loop_potencias`

This screenshot shows the same assembly code as Figure 8, but the CPU Registers window shows updated values after three iterations of the loop. The value in R4 has been multiplied by 4 (from 1099861860 to 1099861864), and the value in R1 has increased from 3 to 7. The assembly code continues to calculate higher powers of 2.

Register	Hex	Interpreted
r0	0x40011044	1073741524
r1	0x7	7
r2	0x0	0
r3	0x9	9
r4	0x41812108	1099861860
r5	0x0	0
r6	0x40000003	1073742817
r7	0x0	0
r8	0x0	0
r9	0x0	0
r10	0x40011000	1073741520
r11	0x0	0
r12	0x41812100	1099861864
sp	0x41812100	1099861864
lr	0x4186c541	1099351457
pc	0x40000000	1073741520 <loop_potencias+24>
cpsr	0x00000010	-214056128
fpscr	0x0	0
fpsid	0x410430f0	1096793712
fpexc	0x40000000	1073741824

Figura 9: 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 shows the 'CPU Registers' window listing registers R0 through R14 with their corresponding memory addresses and values. Below the assembly code, the 'Dspg Editor' window shows assembly mnemonics. At the bottom, the status bar indicates the file path '/home/angel/Desktop/P2/ps2.s' and the current line 'Line 28, Col 1, Pos 1241'.

```

ps2.s
Projects
  P2
    ps2.s
asm
.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	10737411600
R1	0x00000000	3145728
R2	0x14	20
R3	0xfffffd	3145725
R4	0x41011000	10737410800
R5	0x00000000	0
R6	0x40000001	1073742817
R7	0x1	1
R8	0x0	0
R9	0x0	0
R10	0x40011000	10737411406
R11	0x00000000	0
R12	0x41011000	10737410804
R0	0x41011000	10737410804
R1	0x41011000	10737410804
R2	0x41011000	10737410804
R3	0x00000000	0
R4	0x40000001	10737411407
R5	0x40000000	0x40000000 <fin_potencias+1>
R6	0x00000010	3113130256
R7	0x0	0
R8	0x0	0
R9	0x0	0
R10	0x40000000	1073741124

Figura 10: 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 4×3 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 2×2 ; 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 2×2 , 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 reservó 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:

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

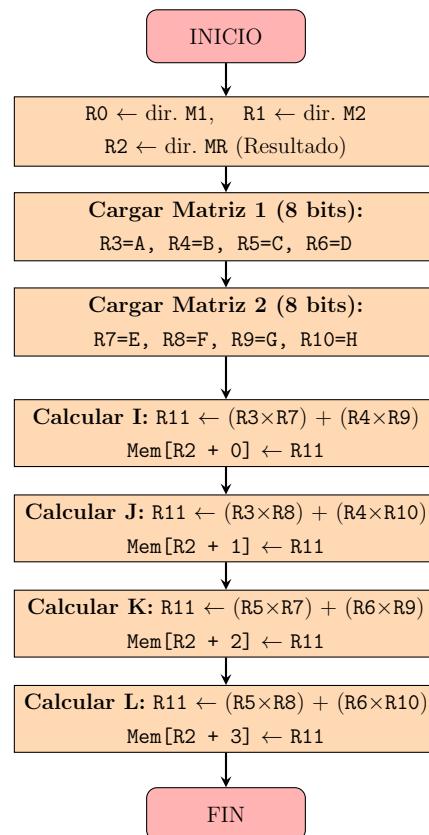
$$J = (A \times F) + (B \times H)$$

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

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

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
- Code Editor:** Displays assembly code for matrix multiplication. The code uses LDR and STR instructions to copy elements from memory into registers (R0-R10) and then performs operations like ADD and SUB to calculate the result matrix (MR).
- Registers View:** Shows the CPU Registers window with the following values:

Register	Hex	Interpreted
r0	0x40011040	1073811520
r1	0x40011041	1073811524
r2	0x40011042	1073811528
r3	0x400004f5	1073743965
r4	0x418121a6	1089891869
r5	0x0	
r6	0x400000e9	1073742825
r7	0x0	
r8	0x0	
r9	0x0	
r10	0x40011040	1073811456
r11	0x40011041	1073811460
r12	0x41812120	1089891864
sp	0x41812190	0x418121216
lr	0x418c5a51	1099321457
pc	0x40000000	0x40000000 <main+8>
cpsr	0x40000010	1074039344
fpscr	0x0	
fpsid	0x410430f0	1096793712
fpexc	0x40000000	1073741824
- Logs & others:** Shows the build log and debugger output. The debugger output indicates it is connected to a remote target and shows the command `At :/home/angel/Descargas/P2/ps2.s:10`.

Figura 11: 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
- Code Editor:** Displays assembly code for matrix multiplication. The code uses LDR and STR instructions to copy elements from memory into registers (R0-R10) and then performs operations like ADD and SUB to calculate the result matrix (MR).
- Registers View:** Shows the CPU Registers window with the following values:

Register	Hex	Interpreted
r0	0x40011040	1073811520
r1	0x40011041	1073811524
r2	0x40011042	1073811528
r3	0x0	
r4	0x1	
r5	0x3	
r6	0x4	
r7	0x0	
r8	0x0	
r9	0x0	
r10	0x40011000	1073811456
r11	0x40011001	1073811457
r12	0x41812120	1089891864
sp	0x41812190	0x418121216
lr	0x418c5a51	1099321457
pc	0x40000010	0x40000010 <main+8>
cpsr	0x40000010	1074039344
fpscr	0x0	
fpsid	0x410430f0	1096793712
fpexc	0x40000000	1073741824
- Logs & others:** Shows the build log and debugger output. The debugger output indicates it is connected to a remote target and shows the command `At :/home/angel/Descargas/P2/ps2.s:10`. It also lists several breakpoints set at various lines of the assembly code.

Figura 12: 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:** Standard icons for file operations.
- Project Explorer:** Projects, Workspace, P2, ASM Sources, ps2.s
- Code Editor:** ps2.s assembly code listing. The code performs matrix multiplication (A * B) + C, accumulating results in R11. It includes comments explaining the purpose of each instruction.
- Registers Window:** CPU Registers table showing the state of registers r0 through r12, sp, lr, pc, cpsr, fpSCR, fpSID, and fpExc.
- Logs & others:** Shows build log output with numerous "At" entries indicating assembly code locations.
- 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 13: 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:** Standard icons for file operations.
- Project Explorer:** Projects, Workspace, P2, ASM Sources, ps2.s
- Code Editor:** ps2.s assembly code listing. The code performs matrix multiplication (A * B) + C, accumulating results in R11. It includes comments explaining the purpose of each instruction.
- Registers Window:** CPU Registers table showing the state of registers r0 through r12, sp, lr, pc, cpsr, fpSCR, fpSID, and fpExc.
- Logs & others:** Shows build log output with numerous "At" entries indicating assembly code locations.
- 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 14: 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
    mul r11, r3, r7      @ R11 = A * E
    mla r11, r4, r9, r11  @ Multiply-Accumulate: R11 = (B * G) + R11
    strb r11, [r2, #0]    @ Guardamos 'I' en la matriz resultado

    # Calculando J = A * F + B * H
    mul r11, r5, r7      @ R11 = A * F
    mla r11, r4, r10, r11 @ R11 = (B * H) + R11
    strb r11, [r2, #1]    @ Guardamos 'J'

    # Calculando K = C * E + D * G
    mul r11, r5, r8      @ R11 = C * E
    mla r11, r6, r10, r11 @ R11 = (D * G) + R11
    strb r11, [r2, #2]    @ Guardamos 'K'

    MOV R7, #1           @ sys_exit
    SVC 8                @ Termina

    # Calculando L = C * F + D * H
    mul r11, r5, r8      @ R11 = C * F
    mla r11, r6, r10, r11 @ R11 = (D * H) + R11
    strb r11, [r2, #3]    @ Guardamos 'L'

```

Register	Hex	Interpreted
R0	0x40011040	1073811230
R1	0x40011044	1073811234
R2	0x40011048	1073811238
R3	0x4001104C	1073811242
R4	0x40011050	1073811244
R5	0x40011054	1073811248
R6	0x40011058	1073811252
R7	0x4001105C	1073811256
R8	0x40011060	1073811260
R9	0x40011064	1073811264
R10	0x40011068	1073811268
R11	0x40011070	1073811270
R12	0x41400000	107381127004
sp	0x41400000	107381127008
r1	0x41400001	107381127047
pc	0x40000000	0x40000000 <main>
spcr	0x40000010	107400004444
r1pcr	0x0	0
r1sr	0x40000000	107381127112
r1sc	0x40000000	107381127124

Figura 15: 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 2×2 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 2×2

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:

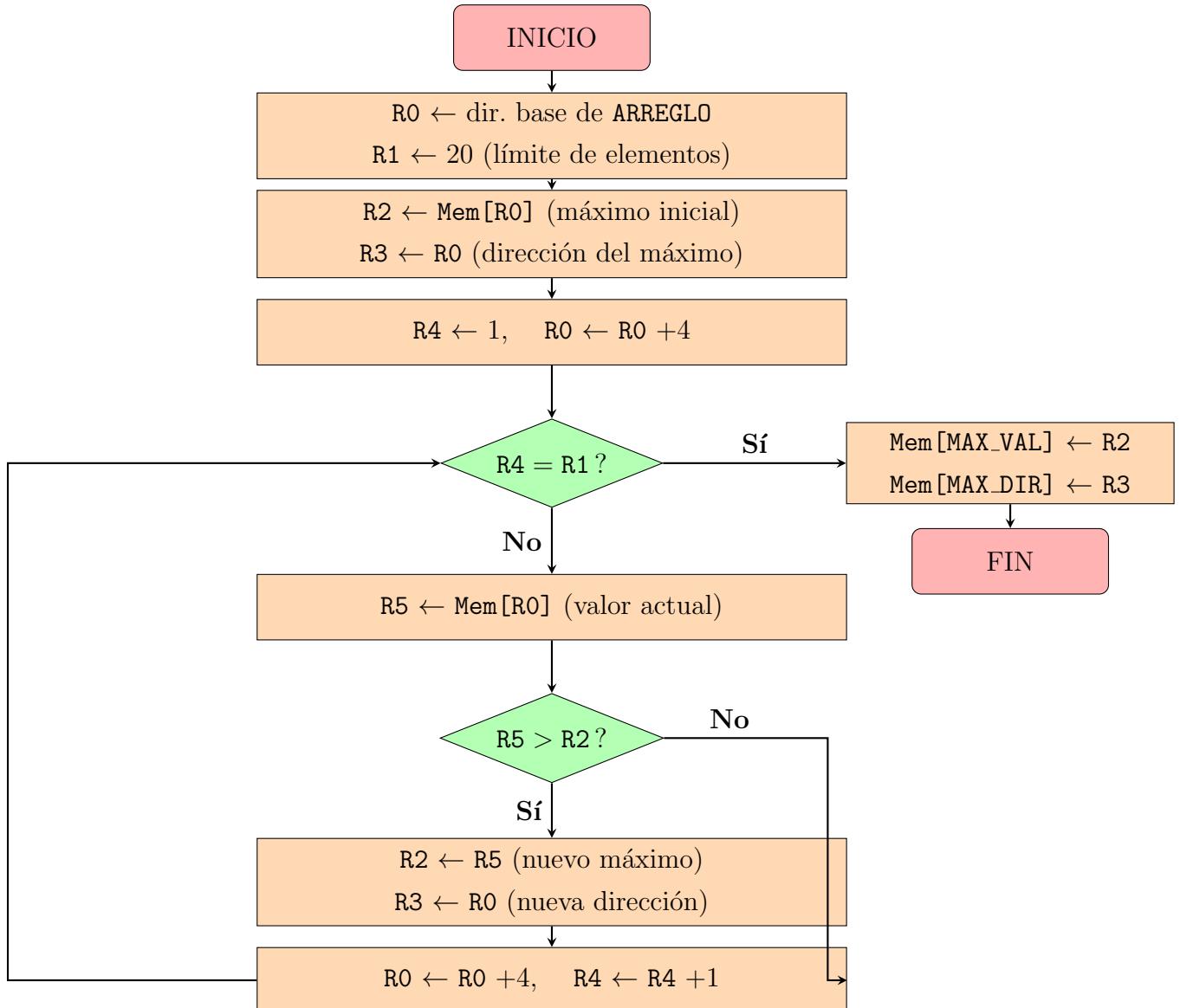
- a) Indicar cuál fue el valor mayor.
- b) Ubicar la dirección donde se encontró este número.
- c) Usar las direcciones que requiera para cumplir lo solicitado.

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. On the left, the project tree shows a workspace with a P2 project containing an ps2.s file. The assembly code in the main window is:

```
.data
    .word ARREGLO, MAX_VAL, MAX_DIR
    ARREGLO: .word 5, 12, 3, 45, 2, 105, 1, 8, 33, 10, 11, 24, 0, 77, 21, 6, 9, 88, 4, 15
    MAX_VAL: .word 0
    MAX_DIR: .word 8
    .text
    .global main
main:
    ldr r0, =ARREGLO           @ R0 = Puntero principal que recorrerá el arreglo
    mov r1, #20                @ R1 = Límite de elementos (20)
    ldr r2, [r0]                @ R2 = Dirección de ARREGLO (Inicia asumiendo que el índice 0 es el mayor)
    mov r3, r0
    mov r4, #1
    add r0, r0, #4              @ Avanzamos el puntero de memoria al índice 1
    ldr r5, [r0]
    cmp r4, r1                @ Compara el contador con 20
    bne fin_búsqueda          @ Si terminamos, salta al final
    ldr r5, [r0]
    cmp r5, r2                @ Compara (Valor_Actual vs Máximo_Registrado)
    bne siguiente              @ Branch if Less or Equal: Si es menor o igual, ignóralo y salta a 'siguiente'
    .end伪指令
fin_búsqueda:
    .end伪指令
siguiente:
    .end伪指令
```

The CPU Registers window on the right shows the initial state of the registers:

Register	Hex	Interpreted
r0	0x40011040	10737811520
r1	0x40011044	10737811524
r2	0x40011048	10737811528
r3	0x2	2
r4	0x1 1	1
r5	0x0 0	0
r6	0x400000009	1073741825
r7	0x0 0	0
r8	0x0 0	0
r9	0x40011000	10737811400
r10	0x0 0	0
r11	0x0 0	0
r12	0x41232108	10900121004
sp	0x41232108	10900121008
lr	0x4156c5ca1	1099515457
pc	0x40000000	104000000 <main+20>
cpsr	0x40000000	1074293544
fpcr	0x0 0	0
fpxid	0x41043070	10907733712
fpxec	0x40000000	1073741824

Figura 16: 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 with the assembly file ps2.s open. The CPU Registers window displays the state of registers: R0=0x40011040, R1=0x40011044, R2=0x40011048, R3=0x69, R4=0x69, R5=0x3, R6=0x4, R7=0x1, R8=0x5, R9=0x2, R10=0x1, R11=0x2, R12=0x4145c000, R9p=0x4145c110, R1r=0x4145c111, R9c=0x40000010, R9pc=0x40000010, R9sr=0x0, R9fd=0x4145c000, R9xc=0x40000000. The assembly code is annotated with comments explaining the logic of finding the maximum value.

Figura 17: 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 with the assembly file ps2.s open. The CPU Registers window displays the state of registers: R0=0x40011040, R1=0x40011044, R2=0x40011048, R3=0x69, R4=0x6, R5=0x6, R6=0x65, R7=0x0, R8=0x0, R9=0x0, R10=0x4145c000, R11=0x4145c110, R12=0x4145c220, R9p=0x40000010, R9pc=0x40000010, R9sr=0x0, R9fd=0x4145c000, R9xc=0x40000000. The assembly code is annotated with comments explaining the logic of finding the maximum value.

Figura 18: El ciclo cursando la iteración 6 ($R4 = 0x6$). El programa ya ha registrado el verdadero número máximo ($0x69$).



ps2.s

```
    ldr r2, [r0]      ; R2 = Guarda el BOCINERO (Inicia asumiendo que el indice 0 es el mayor)
    mov r3, #0        ; R3 = Guarda la DIRECCION del maximo (Inicia con la del indice 0)
    mov r4, #1        ; R4 = Contador de ciclo (inicia en 1 porque ya evaluamos el 0)
    add r0, r0, #4    ; Avanzamos el puntero de memoria al indice 1

buscar_mayor:
    cmp r4, r1      ; Compara el contador con r1
    beq fin_busqueda ; Si son iguales, salta al final
    ldr r5, [r0]      ; R5 = Lee el valor actual de la memoria
    cmp r5, r2      ; Compara Valor_Actual vs Maximo_Registrado
    bne siguiente   ; Branch if Less or Equal: Si es menor o igual, ignóralo y salta a 'siguiente'
    bne llegado     ; Si llegó a esta linea, encontramos un nuevo mayor
    mov r2, r5      ; R2 adopta el nuevo valor mayor
    mov r3, r0      ; R3 adopta la dirección de memoria de este nuevo mayor

siguiente:
    add r0, r0, #4    ; Avanzamos la lectura en la memoria (4 bytes)
    inc r4          ; Incrementamos el contador de ciclo
    b buscar_mayor ; Repetimos

fin_busqueda:
    ldr r6, =MAX_VAL ; Carga dirección para guardar el valor
    str r7, [r1]      ; Almacena en memoria al usuario mayor
```

CPU Registers

Register	Hex	Interpreted
r9	0x40101060	0073811552
r1	0x14	20
r2	0x00	105
r3	0x40400000	0073811548
r4	0x00	0
r5	0x00	0
r6	0x40400030	0073741282
r7	0x00	0
r8	0x00	0
r9	0x00	0
r10	0x40101060	0073811456
r11	0x00	0
r12	0x40101060	0073811540
lr	0x41052100	007352100
pc	0x40400010	0073811518
spcr	0x00000010	-2140561228
fpcr	0x00	0
frsacr	0x40400000	0073811512
fpcx	0x00000000	0073811524

Figura 19: Iteración 8 del ciclo ($R4 = 0x8$). La condición de salto evita que los números menores sobreescriban el valor máximo ya encontrado.

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 structure under "Projects". It shows a "Workspace" containing a "P2" folder, which further contains "ASM Sources" with files "ps2.s" and "ps2_1.asm".

The main editor window shows the assembly code for "ps2.s". The code implements a search algorithm, likely binary search, to find a target value in a sorted array stored in memory. It uses registers R0 through R7, and memory locations like R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, R64, R65, R66, R67, R68, R69, R70, R71, R72, R73, R74, R75, R76, R77, R78, R79, R80, R81, R82, R83, R84, R85, R86, R87, R88, R89, R90, R91, R92, R93, R94, R95, R96, R97, R98, R99, R100, R101, R102, R103, R104, R105, R106, R107, R108, R109, R110, R111, R112, R113, R114, R115, R116, R117, R118, R119, R120, R121, R122, R123, R124, R125, R126, R127, R128, R129, R130, R131, R132, R133, R134, R135, R136, R137, R138, R139, R140, R141, R142, R143, R144, R145, R146, R147, R148, R149, R150, R151, R152, R153, R154, R155, R156, R157, R158, R159, R160, R161, R162, R163, R164, R165, R166, R167, R168, R169, R170, R171, R172, R173, R174, R175, R176, R177, R178, R179, R180, R181, R182, R183, R184, R185, R186, R187, R188, R189, R190, R191, R192, R193, R194, R195, R196, R197, R198, R199, R200, R201, R202, R203, R204, R205, R206, R207, R208, R209, R210, R211, R212, R213, R214, R215, R216, R217, R218, R219, R220, R221, R222, R223, R224, R225, R226, R227, R228, R229, R229, R230, R231, R232, R233, R234, R235, R236, R237, R238, R239, R239, R240, R241, R242, R243, R244, R245, R246, R247, R248, R249, R249, R250, R251, R252, R253, R254, R255, R256, R257, R258, R259, R259, R260, R261, R262, R263, R264, R265, R266, R267, R268, R269, R269, R270, R271, R272, R273, R274, R275, R276, R277, R278, R279, R279, R280, R281, R282, R283, R284, R285, R286, R287, R288, R289, R289, R290, R291, R292, R293, R294, R295, R296, R297, R298, R299, R299, R300, R301, R302, R303, R304, R305, R306, R307, R308, R309, R309, R310, R311, R312, R313, R314, R315, R316, R317, R318, R319, R319, R320, R321, R322, R323, R324, R325, R326, R327, R328, R329, R329, R330, R331, R332, R333, R334, R335, R336, R337, R338, R339, R339, R340, R341, R342, R343, R344, R345, R346, R347, R348, R349, R349, R350, R351, R352, R353, R354, R355, R356, R357, R358, R359, R359, R360, R361, R362, R363, R364, R365, R366, R367, R368, R369, R369, R370, R371, R372, R373, R374, R375, R376, R377, R378, R379, R379, R380, R381, R382, R383, R384, R385, R386, R387, R388, R388, R389, R390, R391, R392, R393, R394, R395, R396, R397, R398, R399, R399, R400, R401, R402, R403, R404, R405, R406, R407, R408, R409, R409, R410, R411, R412, R413, R414, R415, R416, R417, R418, R419, R420, R421, R422, R423, R424, R425, R426, R427, R428, R429, R429, R430, R431, R432, R433, R434, R435, R436, R437, R438, R439, R439, R440, R441, R442, R443, R444, R445, R446, R447, R448, R449, R449, R450, R451, R452, R453, R454, R455, R456, R457, R458, R459, R459, R460, R461, R462, R463, R464, R465, R466, R467, R468, R469, R469, R470, R471, R472, R473, R474, R475, R476, R477, R478, R479, R479, R480, R481, R482, R483, R484, R485, R486, R487, R488, R489, R489, R490, R491, R492, R493, R494, R495, R496, R497, R498, R499, R499, R500, R501, R502, R503, R504, R505, R506, R507, R508, R509, R509, R510, R511, R512, R513, R514, R515, R516, R517, R518, R519, R519, R520, R521, R522, R523, R524, R525, R526, R527, R528, R529, R529, R530, R531, R532, R533, R534, R535, R536, R537, R538, R539, R539, R540, R541, R542, R543, R544, R545, R546, R547, R548, R549, R549, R550, R551, R552, R553, R554, R555, R556, R557, R558, R559, R559, R560, R561, R562, R563, R564, R565, R566, R567, R568, R569, R569, R570, R571, R572, R573, R574, R575, R576, R577, R578, R579, R579, R580, R581, R582, R583, R584, R585, R586, R587, R588, R588, R589, R590, R591, R592, R593, R594, R595, R596, R597, R597, R598, R599, R599, R600, R601, R602, R603, R604, R605, R606, R607, R608, R609, R609, R610, R611, R612, R613, R614, R615, R616, R617, R618, R619, R619, R620, R621, R622, R623, R624, R625, R626, R627, R628, R629, R629, R630, R631, R632, R633, R634, R635, R636, R637, R638, R639, R639, R640, R641, R642, R643, R644, R645, R646, R647, R648, R649, R649, R650, R651, R652, R653, R654, R655, R656, R657, R658, R659, R659, R660, R661, R662, R663, R664, R665, R666, R667, R668, R669, R669, R670, R671, R672, R673, R674, R675, R676, R677, R678, R679, R679, R680, R681, R682, R683, R684, R685, R686, R687, R688, R688, R689, R690, R691, R692, R693, R694, R695, R696, R697, R697, R698, R699, R699, R700, R701, R702, R703, R704, R705, R706, R707, R708, R709, R709, R710, R711, R712, R713, R714, R715, R716, R717, R718, R719, R719, R720, R721, R722, R723, R724, R725, R726, R727, R728, R729, R729, R730, R731, R732, R733, R734, R735, R736, R737, R738, R739, R739, R740, R741, R742, R743, R744, R745, R746, R747, R748, R749, R749, R750, R751, R752, R753, R754, R755, R756, R757, R758, R759, R759, R760, R761, R762, R763, R764, R765, R766, R767, R768, R769, R769, R770, R771, R772, R773, R774, R775, R776, R777, R778, R779, R779, R780, R781, R782, R783, R784, R785, R786, R787, R788, R788, R789, R790, R791, R792, R793, R794, R795, R796, R797, R797, R798, R799, R799, R800, R801, R802, R803, R804, R805, R806, R807, R808, R809, R809, R810, R811, R812, R813, R814, R815, R816, R817, R818, R819, R819, R820, R821, R822, R823, R824, R825, R826, R827, R828, R829, R829, R830, R831, R832, R833, R834, R835, R836, R837, R838, R839, R839, R840, R841, R842, R843, R844, R845, R846, R847, R848, R849, R849, R850, R851, R852, R853, R854, R855, R856, R857, R858, R859, R859, R860, R861, R862, R863, R864, R865, R866, R867, R868, R869, R869, R870, R871, R872, R873, R874, R875, R876, R877, R878, R879, R879, R880, R881, R882, R883, R884, R885, R886, R887, R888, R889, R889, R890, R891, R892, R893, R894, R895, R896, R897, R897, R898, R899, R899, R900, R901, R902, R903, R904, R905, R906, R907, R908, R909, R909, R910, R911, R912, R913, R914, R915, R916, R917, R918, R919, R919, R920, R921, R922, R923, R924, R925, R926, R927, R928, R929, R929, R930, R931, R932, R933, R934, R935, R936, R937, R938, R939, R939, R940, R941, R942, R943, R944, R945, R946, R947, R948, R949, R949, R950, R951, R952, R953, R954, R955, R956, R957, R958, R959, R959, R960, R961, R962, R963, R964, R965, R966, R967, R968, R969, R969, R970, R971, R972, R973, R974, R975, R976, R977, R978, R979, R979, R980, R981, R982, R983, R984, R985, R986, R987, R988, R988, R989, R989, R990, R991, R992, R993, R994, R995, R996, R997, R997, R998, R998, R999, R999, R1000, R1001, R1002, R1003, R1004, R1005, R1006, R1007, R1008, R1009, R1009, R1010, R1011, R1012, R1013, R1014, R1015, R1016, R1017, R1018, R1019, R1019, R1020, R1021, R1022, R1023, R1024, R1025, R1026, R1027, R1028, R1029, R1029, R1030, R1031, R1032, R1033, R1034, R1035, R1036, R1037, R1038, R1039, R1039, R1040, R1041, R1042, R1043, R1044, R1045, R1046, R1047, R1048, R1049, R1049, R1050, R1051, R1052, R1053, R1054, R1055, R1056, R1057, R1058, R1059, R1059, R1060, R1061, R1062, R1063, R1064, R1065, R1066, R1067, R1068, R1069, R1069, R1070, R1071, R1072, R1073, R1074, R1075, R1076, R1077, R1078, R1079, R1079, R1080, R1081, R1082, R1083, R1084, R1085, R1086, R1087, R1087, R1088, R1089, R1089, R1090, R1091, R1092, R1093, R1094, R1095, R1096, R1097, R1097, R1098, R1099, R1099, R1100, R1101, R1102, R1103, R1104, R1105, R1106, R1107, R1108, R1109, R1109, R1110, R1111, R1112, R1113, R1114, R1115, R1116, R1117, R1118, R1119, R1119, R1120, R1121, R1122, R1123, R1124, R1125, R1126, R1127, R1128, R1129, R1129, R1130, R1131, R1132, R1133, R1134, R1135, R1136, R1137, R1138, R1139, R1139, R1140, R1141, R1142, R1143, R1144, R1145, R1146, R1147, R1148, R1149, R1149, R1150, R1151, R1152, R1153, R1154, R1155, R1156, R1157, R1158, R1159, R1159, R1160, R1161, R1162, R1163, R1164, R1165, R1166, R1167, R1168, R1169, R1169, R1170, R1171, R1172, R1173, R1174, R1175, R1176, R1177, R1178, R1179, R1179, R1180, R1181, R1182, R1183, R1184, R1185, R1186, R1187, R1187, R1188, R1189, R1189, R1190, R1191, R1192, R1193, R1194, R1195, R1196, R1197, R1197, R1198, R1199, R1199, R1200, R1201, R1202, R1203, R1204, R1205, R1206, R1207, R1208, R1209, R1209, R1210, R1211, R1212, R1213, R1214, R1215, R1216, R1217, R1218, R1219, R1219, R1220, R1221, R1222, R1223, R1224, R1225, R1226, R1227, R1228, R1229, R1229, R1230, R1231, R1232, R1233, R1234, R1235, R1236, R1237, R1238, R1239, R1239, R1240, R1241, R1242, R1243, R1244, R1245, R1246, R1247, R1248, R1249, R1249, R1250, R1251, R1252, R1253, R1254, R1255, R1256, R1257, R1258, R1259, R1259, R1260, R1261, R1262, R1263, R1264, R1265, R1266, R1267, R1268, R1269, R1269, R1270, R1271, R1272, R1273, R1274, R1275, R1276, R1277, R1278, R1279, R1279, R1280, R1281, R1282, R1283, R1284, R1285, R1286, R1287, R1287, R1288, R1289, R1289, R1290, R1291, R1292, R1293, R1294, R1295, R1296, R1297, R1297, R1298, R1299, R1299, R1300, R1301, R1302, R1303, R1304, R1305, R1306, R1307, R1308, R1309, R1309, R1310, R1311, R1312, R1313, R1314, R1315, R1316, R1317, R1318, R1319, R1319, R1320, R1321, R1322, R1323, R1324, R1325, R1326, R1327, R1328, R1329, R1329, R1330, R1331, R1332, R1333, R1334, R1335, R1336, R1337, R1338, R1339, R1339, R1340, R1341, R1342, R1343, R1344, R1345, R1346, R1347, R1348, R1349, R1349, R1350, R1351, R1352, R1353, R1354, R1355, R1356, R1357, R1358, R1359, R1359, R1360, R1361, R1362, R1363, R1364, R1365, R1366, R1367, R1368, R1369, R1369, R1370, R1371, R1372, R1373, R1374, R1375, R1376, R1377, R1378, R1379, R1379, R1380, R1381, R1382, R1383, R1384, R1385, R1386, R1387, R1387, R1388, R1389, R1389, R1390, R1391, R1392, R1393, R1394, R1395, R1396, R1397, R1397, R1398, R1399, R1399, R1400, R1401, R1402, R1403, R1404, R1405, R1406, R1407, R1408, R1409, R1409, R1410, R1411, R1412, R1413, R1414, R1415, R1416, R1417, R1418, R1419, R1419, R1420, R1421, R1422, R1423, R1424, R1425, R1426, R1427, R1428, R1429, R1429, R1430, R1431, R1432, R1433, R1434, R1435, R1436, R1437, R1438, R1439, R1439, R1440, R1441, R1442, R1443, R1444, R1445, R1446, R1447, R1448, R1449, R1449, R1450, R1451, R1452, R1453, R1454, R1455, R1456, R1457, R1458, R1459, R1459, R1460, R1461, R1462, R1463, R1464, R1465, R1466, R1467, R1468, R1469, R1469, R1470, R1471, R1472, R1473, R1474, R1475, R1476, R1477, R1478, R1479, R1479, R1480, R1481, R1482, R1483, R1484, R1485, R1486, R1487, R1488, R1488, R1489, R1489, R1490, R1491, R1492, R1493, R1494, R1495, R1496, R1497, R1497, R1498, R1499, R1499, R1500, R1501, R1502, R1503, R1504, R1505, R1506, R1507, R1508, R1509, R1509, R1510, R1511, R1512, R1513, R1514, R1515, R1516, R1517, R1518, R1519, R1519, R1520, R1521, R1522, R1523, R1524, R1525, R1526, R1527, R1528, R1529, R1529, R1530, R1531, R1532, R1533, R1534, R1535, R1536, R1537, R1538, R1539, R1539, R1540, R1541, R1542, R1543, R1544, R1545, R1546, R1547, R1548, R1549, R1549, R1550, R1551, R1552, R1553, R1554, R1555, R1556, R1557, R1558, R1559, R1559, R1560, R1561, R1562, R1563, R1564, R1565, R1566, R1567, R1568, R1569, R1569, R1570, R1571, R1572, R1573, R1574, R1575, R1576, R1577, R1578, R1579, R1579, R1580, R1581, R1582, R1583, R1584, R1585, R1586, R1587, R1587, R1588, R1589, R1589, R1590, R1591, R1592, R1593, R1594, R1595, R1596, R1597, R1597, R1598, R1599, R1599, R1600, R1601, R1602, R1603, R1604, R1605, R1606, R1607, R1608, R1609, R1609, R1610, R1611, R1612, R1613, R1614, R1615, R1616, R1617, R1618, R1619, R1619, R1620, R1621, R1622, R1623, R1624, R1625, R1626, R1627, R1628, R1629, R1629, R1630, R1631, R1632, R1633, R1634, R1635, R1636, R1637, R1638, R1639, R1639, R1640, R1641, R1642, R1643, R1644, R1645, R1646, R1647, R1648, R1649, R1649, R1650, R1651, R1652, R1653, R1654, R1655, R1656, R1657, R1658, R1659, R1659, R1660, R1661, R1662, R1663, R1664, R1665, R1666, R1667, R1668, R1669, R1669, R1670, R1671, R1672, R1673, R1674, R1675, R1676, R1677, R1678, R1679, R1679, R1680, R1681, R1682, R1683, R1684, R1685, R1686, R1687, R1687, R1688, R1689, R1689, R1690, R1691, R1692, R1693, R1694, R1695, R1696, R1697, R1697, R1698, R1699, R1699, R1700, R1701, R1702, R1703, R1704, R1705, R1706, R1707, R1708, R1709, R1709, R1710, R1711, R1712, R1713, R1714, R1715, R1716, R1717, R1718, R1719, R1719, R1720, R1721, R1722, R1723, R1724, R1725, R1726, R1727, R1728, R1729, R1729, R1730, R1731, R1732, R1733, R1734, R1735, R1736, R1737, R1738, R1739, R1739, R1740, R1741, R1742, R1743, R1744, R1745, R1746, R1747, R1748, R1749, R1749, R1750, R1751, R1752, R1753, R1754, R1755, R1756, R1757, R1758, R1759, R1759, R1760, R1761, R1762, R1763, R1764, R1765, R1766, R1767, R1768, R1769, R1769, R1770, R1771, R1772, R1773, R1774, R1775, R1776, R1777, R1778, R1779, R1779, R1780, R1781, R1782, R1783, R1784, R1785, R1786, R1787, R1787, R1788, R1789, R1789, R1790, R1791, R1792, R1793, R1794, R1795, R1796, R1797, R1797, R1798, R1799, R1799, R1800, R1801, R1802, R1803, R1804, R1805, R1806, R1807, R1808, R1809, R1809, R1810, R1811, R1812, R1813, R1814, R1815, R1816, R1817, R1818, R1819, R1819, R1820, R1821, R1822, R1823, R1824, R1825, R1826, R1827, R1828, R1829, R1829, R1830, R1831, R1832, R1833, R1834, R1835, R1836, R1837, R1838, R1839, R1839, R1840, R1841, R1842, R1843, R1844, R1845, R1846, R1847, R1848, R1849, R1849, R1850, R1851, R1852, R1853, R1854, R1855, R1856, R1857, R1858, R1859, R1859, R1860, R1861, R1862, R1863, R1864, R1865, R1866, R1867, R1868, R1869, R1869, R1870, R1871, R1872, R1873, R1874, R1875, R1876, R1877, R1878, R1879, R1879, R1880, R1881, R1882, R1883, R1884, R1885, R1886, R1887, R1887, R1888, R1889, R1889, R1890, R1891, R1892, R1893, R1894, R1895, R1896, R1897, R1897, R1898, R1899, R1899, R1900, R1901, R1902, R1903, R1904, R1905, R1906, R1907, R1908, R1909, R1909, R1910, R1911, R1912, R1913, R1914, R1915, R1916, R1917, R1918, R1919, R1919, R1920, R1921, R1922, R1923, R1924, R1925, R1926, R1927, R1928, R1929, R1929, R1930, R1931, R1932, R1933, R1934, R1935, R1936, R1937, R1938, R1939, R1939, R1940, R1941, R1942, R1943, R1944, R1945, R1946, R1947, R1948, R1949, R1949, R1950, R1951, R1952, R1953, R1954, R1955, R1956, R1957, R1958, R1959, R1959, R1960, R1961, R1962, R1963, R1964, R1965, R1966, R1967, R1968, R1969, R1969, R1970, R1971, R1972, R1973, R1974, R1975, R1976, R1977, R1978, R1979, R1979, R1980, R1981, R1982, R1983, R1984, R1985, R1986, R1987, R1987, R1988, R1989, R1989, R1990, R1991, R1992, R1993, R1994, R1995, R1996, R1997, R1997, R1998, R1999, R1999, R2000, R2001, R2002, R2003, R2004, R2005, R2006, R2007, R2008, R2009, R2009, R2010, R2011, R2012, R2013, R2014, R2015, R2016, R2017, R2018, R2

Figura 20: 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	$A[31]$
--------	--------	--------	----------	---------

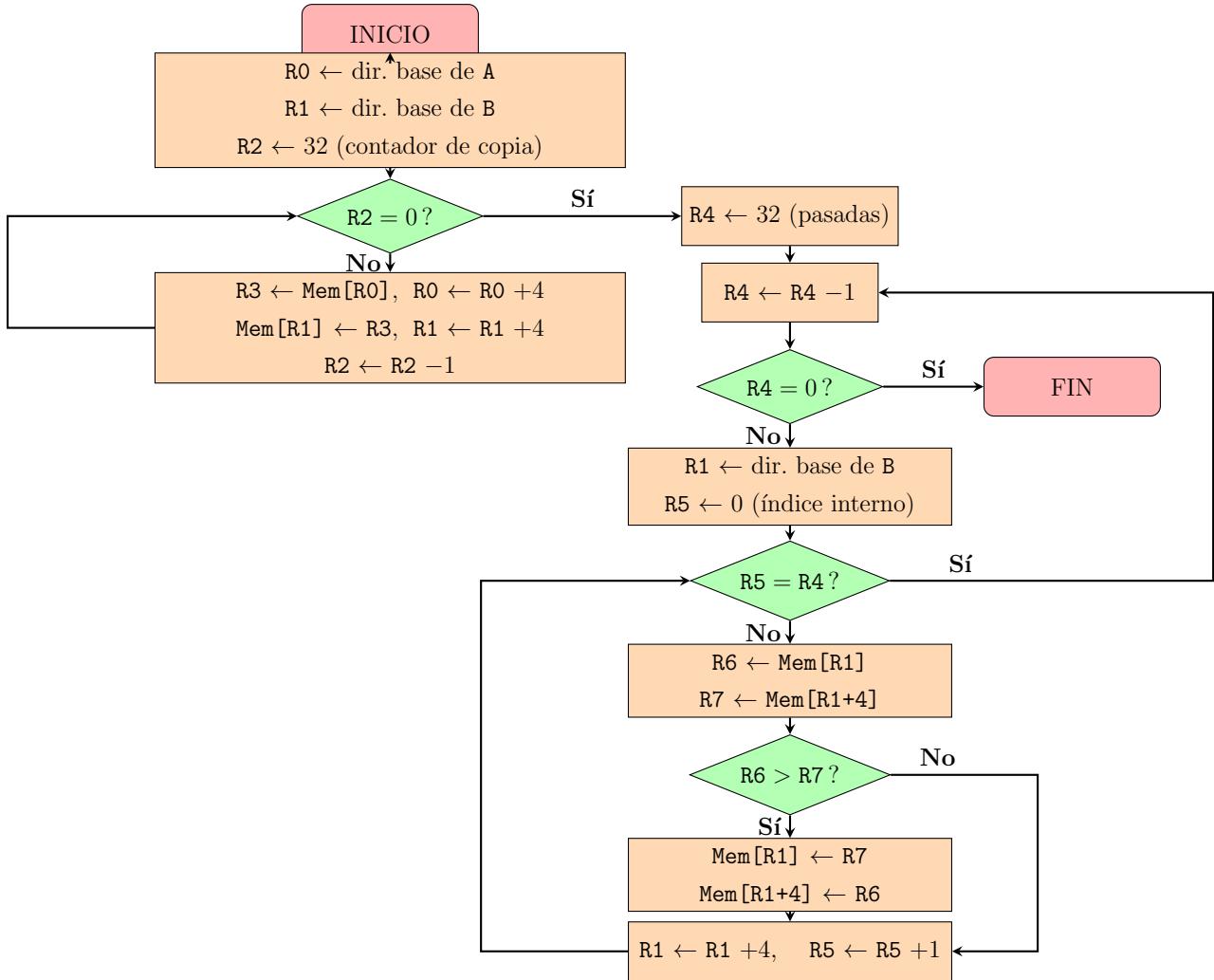
Arreglo ordenado.

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 R_0 (origen en A) y R_1 (destino en B) junto con el contador R_2 con valor 32. El ciclo de copia evalúa si R_2 ha llegado a cero; mientras no lo sea, se lee un dato de A mediante post-indexado (incrementando R_0), se escribe en B (incrementando R_1), y se decrementa el contador, repitiendo el proceso. Cuando R_2 alcanza cero, los 32 elementos han sido transferidos y el flujo pasa a la Fase 2.

En la **Fase 2** se inicializa R_4 con 32 para controlar las pasadas del algoritmo burbuja. Al inicio de cada pasada se decrementa R_4 ; si llega a cero, el ordenamiento está completo y el programa termina. En caso contrario, se reinicializa el apuntador R_1 a la base de B y el índice interno R_5 a cero. El bucle interno compara R_5 con R_4 : 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

```

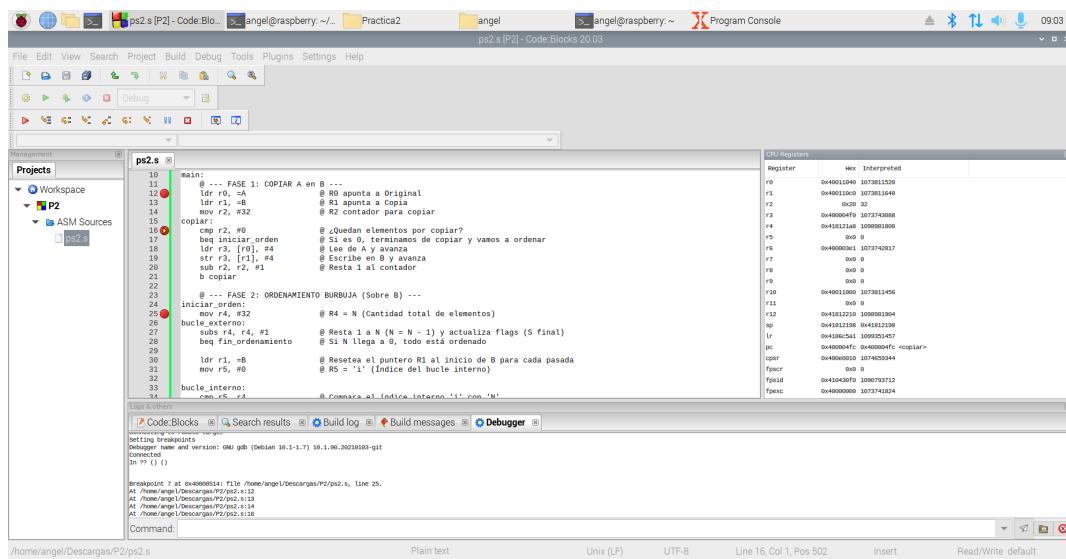


Figura 21: Fase 1 (Copiado). Inicialización de los apuntadores base para el arreglo original (R0) y el arreglo copia (R1).



The screenshot shows the Code::Blocks IDE interface. The assembly code for file ps2.s is displayed in the main window. The CPU Registers window shows the state of registers R0 through R13. The assembly code includes comments explaining the steps of the copy algorithm, such as 'COPIAR A en B' and 'ORDENAMIENTO BURBUJA (Sobre B)'. The debugger window at the bottom shows the current memory dump and command line.

```
ps2.s
main: ... FASE 1: COPIAR A en B ...
    ldr r0, =A          @ R0 apunta a Original
    ldr r1, =B          @ R1 apunta a Copia
    mov r2, #32         @ R2 contador para copiar
    copiar:             @ Quedan elementos por copiar?
    cmp r2, #0          @ Si es 0, terminamos de copiar y vamos a ordenar
    bne copiar, orden, @ Lee de A y avanza
    ldr r3, [r0], #4     @ Lee de A y avanza
    str r3, [r1], #4     @ Escribe en B y avanza
    sub r2, #1           @ Resta 1 al contador
    b copiar             @ Resta 1 al contador
...
    @ ... FASE 2: ORDENAMIENTO BURBUJA (Sobre B) ...
    iniciar_orden:      @ Iniciamos ordenamiento
    mov r4, #32         @ R4 = N (Cantidad total de elementos)
    bucle_externo:      @ Bucle externo
    subs r4, r4, #1     @ Resta 1 a N (N - 1) y actualiza flags (S Final)
    beq fin_ordenamiento, @ Si N llega a 0, todo està ordenado
    ldr r1, =B          @ Resta 1 a N (N - 1) y actualiza flags (S Final)
    mov r5, #0          @ Resetea el puntero R1 al inicio de B para cada pasada
    bucle_interno:      @ Bucle interno
    cmn r5, r4           @ Compara el indice interno 'i' con 'N'
...
    @ ... FASE 2: ORDENAMIENTO BURBUJA (Sobre B) ...
    iniciar_orden:      @ Iniciamos ordenamiento
    mov r4, #32         @ R4 = N (Cantidad total de elementos)
    bucle_externo:      @ Bucle externo
    subs r4, r4, #1     @ Resta 1 a N (N - 1) y actualiza flags (S Final)
    beq fin_ordenamiento, @ Si N llega a 0, todo està ordenado
    ldr r1, =B          @ Resta 1 a N (N - 1) y actualiza flags (S Final)
    mov r5, #0          @ Resetea el puntero R1 al inicio de B para cada pasada
    bucle_interno:      @ Bucle interno
    cmn r5, r4           @ Compara el indice interno 'i' con 'N'

CPU Registers
```

Figura 22: Primera iteración del ciclo de copiado. El registro R3 extrae el primer valor a transferir.

The screenshot shows the Code::Blocks IDE interface. The assembly code for file ps2.s is displayed in the main window. The CPU Registers window shows the state of registers R0 through R13. The assembly code includes comments explaining the steps of the copy algorithm, such as 'COPIAR A en B' and 'ORDENAMIENTO BURBUJA (Sobre B)'. The debugger window at the bottom shows the current memory dump and command line.

```
ps2.s
main: ... FASE 1: COPIAR A en B ...
    ldr r0, =A          @ R0 apunta a Original
    ldr r1, =B          @ R1 apunta a Copia
    mov r2, #32         @ R2 contador para copiar
    copiar:             @ Quedan elementos por copiar?
    cmp r2, #0          @ Si es 0, terminamos de copiar y vamos a ordenar
    bne copiar, orden, @ Lee de A y avanza
    ldr r3, [r0], #4     @ Lee de A y avanza
    str r3, [r1], #4     @ Escribe en B y avanza
    sub r2, #1           @ Resta 1 al contador
    b copiar             @ Resta 1 al contador
...
    @ ... FASE 2: ORDENAMIENTO BURBUJA (Sobre B) ...
    iniciar_orden:      @ Iniciamos ordenamiento
    mov r4, #32         @ R4 = N (Cantidad total de elementos)
    bucle_externo:      @ Bucle externo
    subs r4, r4, #1     @ Resta 1 a N (N - 1) y actualiza flags (S Final)
    beq fin_ordenamiento, @ Si N llega a 0, todo està ordenado
    ldr r1, =B          @ Resta 1 a N (N - 1) y actualiza flags (S Final)
    mov r5, #0          @ Resetea el puntero R1 al inicio de B para cada pasada
    bucle_interno:      @ Bucle interno
    cmn r5, r4           @ Compara el indice interno 'i' con 'N'
...
    @ ... FASE 2: ORDENAMIENTO BURBUJA (Sobre B) ...
    iniciar_orden:      @ Iniciamos ordenamiento
    mov r4, #32         @ R4 = N (Cantidad total de elementos)
    bucle_externo:      @ Bucle externo
    subs r4, r4, #1     @ Resta 1 a N (N - 1) y actualiza flags (S Final)
    beq fin_ordenamiento, @ Si N llega a 0, todo està ordenado
    ldr r1, =B          @ Resta 1 a N (N - 1) y actualiza flags (S Final)
    mov r5, #0          @ Resetea el puntero R1 al inicio de B para cada pasada
    bucle_interno:      @ Bucle interno
    cmn r5, r4           @ Compara el indice interno 'i' con 'N'

CPU Registers
```

Figura 23: 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. The right pane shows the CPU Registers window, listing registers R0 through R11, their memory addresses, and their current values. The assembly code includes comments explaining the steps of the bubble sort algorithm, such as copying elements from array A to array B and then ordering them.

```
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     mov r3, r2, #32
15     copiar:             @ Quedan elementos por copiar?
16         cmp r2, #0        @ Si es cero terminamos de copiar y vamos a ordenar
17         bne fin_ordenado
18         ldr r3, [r0], #4    @ Lee de A y avanza
19         str r3, [r1], #4    @ Escribe en B y avanza
20         add r0, r0, #1
21         sub r1, r1, #1
22         b copiar
23
24     ... FASE 2: ORDENAMIENTO BURBUJA (Sobre B) ...
25     inicial_orden:       @ Inicializa R4, #32      @ R4 = N (Cantidad total de elementos)
26     bucle_externo:       @ Bucle que termina cuando R4 sea cero
27     ldr r4, r4, #1        @ Resta 1 a N (N - 1) y actualiza flags (S final)
28     beq fin_ordenamiento
29     ldr r1, =B
30     mov r5, #0
31     mov r6, r1
32     sub r2, r2, #1
33     b copiar
34     bucle_interno:       @ Compara el indice interno 'i' con 'N'
35     cmp r5, r4
36     bne bucle_externo
37     ldr r2, =R1
38     ldr r1, =R1_Value_actuall
```

Register	Hex	Interpreted
R0	0x40011000	1073811648
R1	0x40011100	1073811776
R2	0x00 0	
R3	0x01 1	
R4	0x41011000	1073811800
R5	0x00 0	
R6	0x40000000	1073742817
R7	0x00 0	
R8	0x00 0	
R9	0x00 0	
R10	0x40011000	1073811456
R11	0x41011000	1073811204
R12	0x41011000	1073811204
sp	0x415e2100	1073812108
lr	0x415e0101	1073811407
pc	0x40000514	1073810514 <iniciar_orden>
rper	0x00000010	1073810326
fpacr	0x00 0	
fpcsr	0x00000000	1073810372
fpcid	0x40000000	1073741024
fpcsc	0x40000000	1073741024

Figura 24: 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. The right pane shows the CPU Registers window, listing registers R0 through R11, their memory addresses, and their current values. The assembly code includes comments explaining the steps of the bubble sort algorithm, such as copying elements from array A to array B and then ordering them.

```
ps2.s
13     ldr r1, =B          @ R1 apunta a Copia
14     mov r2, #32         @ R2 contador para copiar
15     copiar:             @ Quedan elementos por copiar?
16         cmp r2, #0        @ Si es cero terminamos de copiar y vamos a ordenar
17         bne fin_ordenado
18         ldr r3, [r0], #4    @ Lee de A y avanza
19         str r3, [r1], #4    @ Escribe en B y avanza
20         sub r2, r2, #1
21         b copiar
22
23     ... FASE 2: ORDENAMIENTO BURBUJA (Sobre B) ...
24     inicial_orden:       @ Inicializa R4, #32      @ R4 = N (Cantidad total de elementos)
25     bucle_externo:       @ Bucle que termina cuando R4 sea cero
26     ldr r4, r4, #1        @ Resta 1 a N (N - 1) y actualiza flags (S final)
27     beq fin_ordenamiento
28     ldr r1, =B
29     mov r5, #0
30     mov r6, r1
31     sub r2, r2, #1
32     b copiar
33     bucle_interno:       @ Compara el indice interno 'i' con 'N'
34     cmp r5, r4
35     bne bucle_externo
36     ldr r2, =R1
37     ldr r1, =R1_Value_actuall
```

Register	Hex	Interpreted
R0	0x40011000	1073811648
R1	0x40011100	1073811776
R2	0x00 0	
R3	0x01 1	
R4	0x41011000	1073811456
R5	0x00 0	
R6	0x00 0	
R7	0x00 0	
R8	0x00 0	
R9	0x00 0	
R10	0x40000000	1073811456
R11	0x41011000	1073811204
R12	0x41011000	1073811204
sp	0x415e2100	1073812108
lr	0x415e0101	1073811407
pc	0x40000514	1073810514 <iniciar_orden>
rper	0x00000010	1073810326
fpacr	0x00 0	
fpcsr	0x00000000	1073810372
fpcid	0x40000000	1073741024
fpcsc	0x40000000	1073741024

Figura 25: 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:

- Project:** ps2.s [P2] - Code.Blo.
- File:** ps2.s
- Registers:** CPU Registers table showing register values for R0 to R13.
- Code:** Assembly code for the swap operation, including comments explaining the logic of comparing adjacent elements R6 and R7.
- Log:** Shows build log messages.
- Status Bar:** Displays file path (/home/angel/Desktop/P2/ps2.s), line number (Line 38, Col 1, Pos 1510), and encoding (Plain text).

Figura 26: Carga de elementos adyacentes en R6 y R7 para su respectiva comparación.

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 for R0 to R13.
- Code:** Assembly code for the swap operation, including comments explaining the logic of comparing adjacent elements R6 and R7.
- Log:** Shows build log messages.
- Status Bar:** Displays file path (/home/angel/Desktop/P2/ps2.s), line number (Line 44, Col 1, Pos 1838), and encoding (Plain text).

Figura 27: 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 the assembly code for "ps2.s". The code implements a bubble sort algorithm with comments explaining the logic. It includes labels like "bucle_interno:", "bucle_externo:", and "no_cambiar:". The assembly instructions are annotated with comments such as "Resetea el puntero R1 al inicio de B para cada pasada" and "Comparas si el actual es mayor que el derecho".
- Registers Window:** Shows the CPU Registers for the ARM processor. The registers listed are r0 through r15, plus lr, pc, sp, fp, and fpcr. Each register has a hex value and an interpreted value.
- Code Block View:** Shows the assembly code with line numbers and labels.
- Status Bar:** At the bottom, it shows the file path "/home/angel/Desktop/practicas/practica2/practica2.s", the current mode "Plain text", and the line number "Line 48, Col 1, Pos 2039".

Figura 28: 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	0x401000f0	1080733712
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
Detected OS: Linux
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.
Temporary breakpoint 37 at pc 0x4000004f.
Temporary breakpoint 38 at pc 0x4000004f.
Temporary breakpoint 39 at pc 0x4000004f.
Temporary breakpoint 40 at pc 0x4000004f.
Temporary breakpoint 41 at pc 0x4000004f.
Temporary breakpoint 42 at pc 0x4000004f.
Temporary breakpoint 43 at pc 0x4000004f.
Temporary breakpoint 44 at pc 0x4000004f.
Temporary breakpoint 45 at pc 0x4000004f.
Temporary breakpoint 46 at pc 0x4000004f.
Temporary breakpoint 47 at pc 0x4000004f.
Temporary breakpoint 48 at pc 0x4000004f.
Temporary breakpoint 49 at pc 0x4000004f.
Temporary breakpoint 50 at pc 0x4000004f.
Temporary breakpoint 51 at pc 0x4000004f.
Temporary breakpoint 52 at pc 0x4000004f.
Temporary breakpoint 53 at pc 0x4000004f.
Temporary breakpoint 54 at pc 0x4000004f.
```

The Command window shows the command: `run`.

Figura 29: 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 32×4).

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 reinitializa 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:**
- **Lara Hernandez Angel Husiel:**



Referencias

- Anaya, R. (s.f.). *Manual de recursos y aplicaciones Plataforma Raspberry Pi*. <https://odin.fi-b.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.^a 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>