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

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

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

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

loop:
    cmp r2, #16                 @ Compara el contador R2 con el límite de 16
```

```
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

```
/* ACTIVIDAD 2: Direccionamiento Post-indexado con 32 datos
   Objetivo: Guardar 32 números usando auto-incremento de dirección.
*/
.data
    i: .skip 128              @ Reserva 128 bytes (32 elementos * 4
       bytes cada uno)

.text
.global main

main:
    ldr r1, =i                @ Carga en R1 la dirección base del
       arreglo 'i'
    mov r2, #0                @ R2 es el contador, inicia en 0

loop2:
    cmp r2, #32               @ Compara el contador con 32 (el doble
        que la act. 1)
    beq salir                 @ Si llegamos a 32, salta a 'salir'

    str r2, [r1], #4          @ DIRECCIONAMIENTO POST-INDEXADO:
       Guarda R2 en la memoria de R1, y LUEGO suma 4 a R1 automá
       ticamente.
    add r2, r2, #1            @ Incrementa el contador R2 en 1
    b loop2                   @ Repite el bucle

salir:
```

```
23      MOV R7, #1                  @ Prepara sys_exit
24      SVC 0                       @ Termina ejecución
```

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.

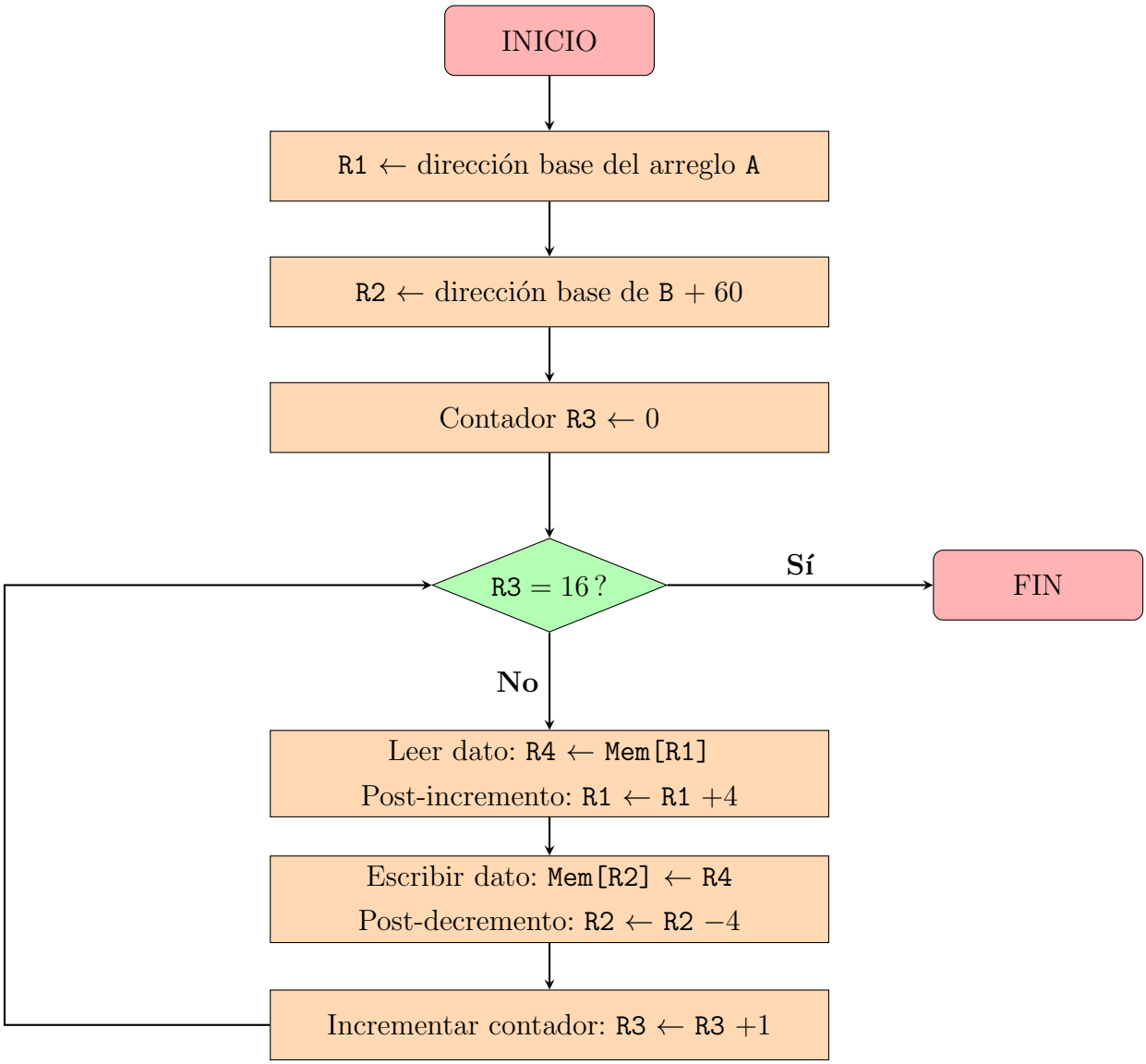
$$A = [\,\text{dato}_1,\ \text{dato}_2,\ \text{dato}_3,\ \text{dato}_4,\ \ldots,\ \text{dato}_{15},\ \text{dato}_{16}\,] \quad \text{@Original}$$

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

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 espejeo 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

```
/* ACTIVIDAD 3: Copia de arreglo invertida
   Objetivo: A = [1..16], B = [16..1]
*/
.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 ÚLTIMO
        espacio de 'B' (15 posiciones * 4 bytes = +60)
    mov r3, #0                 @ R3 es el contador, inicia en 0

loop_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 ATRÁS (-4 bytes)

    add r3, r3, #1             @ Aumenta el contador de copiados
    b loop_copia               @ Repite el ciclo

```

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

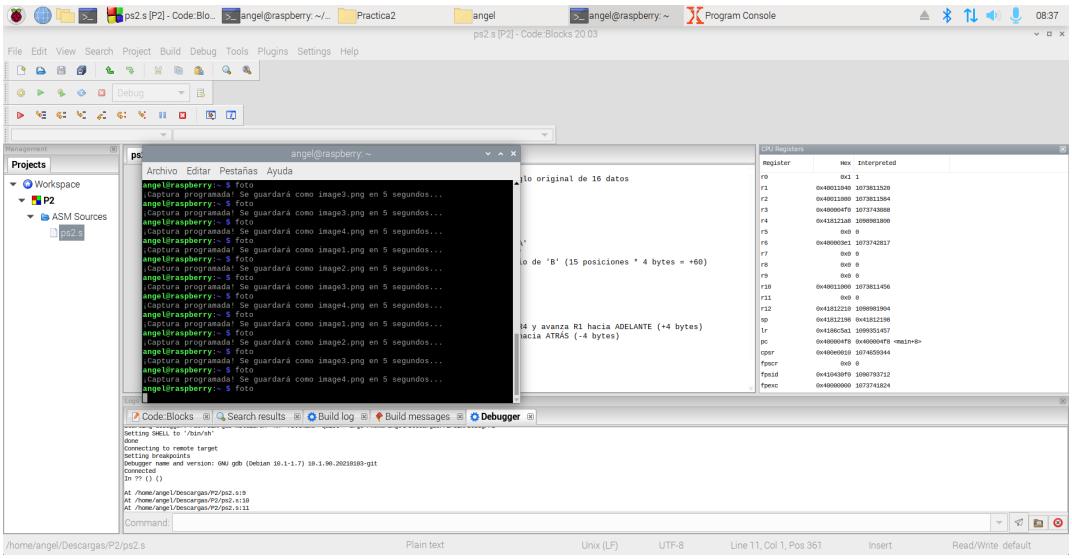


Figura 1: Entorno Code::Blocks preparado para iniciar la depuración del código fuente.

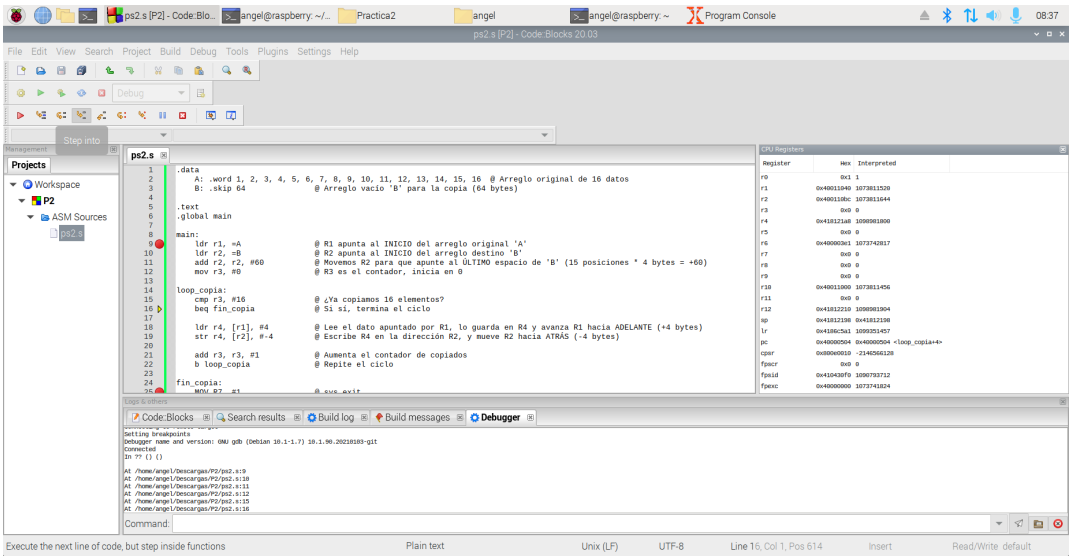


Figura 2: Inicialización de apuntadores. Se observa que `R1` apunta al inicio de A (`0x40011040`) y `R2` apunta al final de B (`0x400110bc`). El contador `R3` inicia en 0.

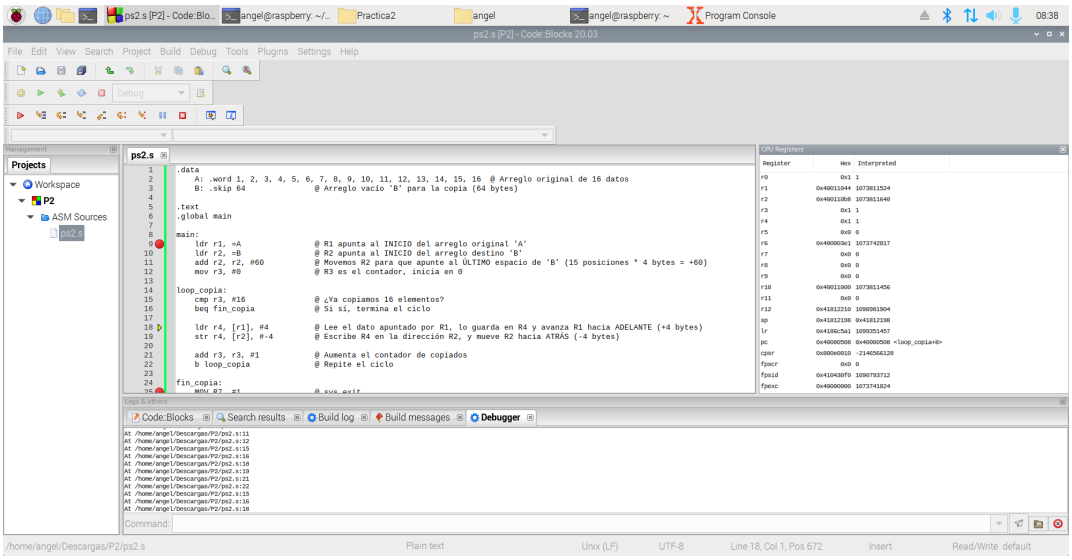


Figura 3: Evaluación de la condición de salida (`CMP r3, #16`) en las primeras fases del ciclo. La ejecución entra al bloque de copiado.

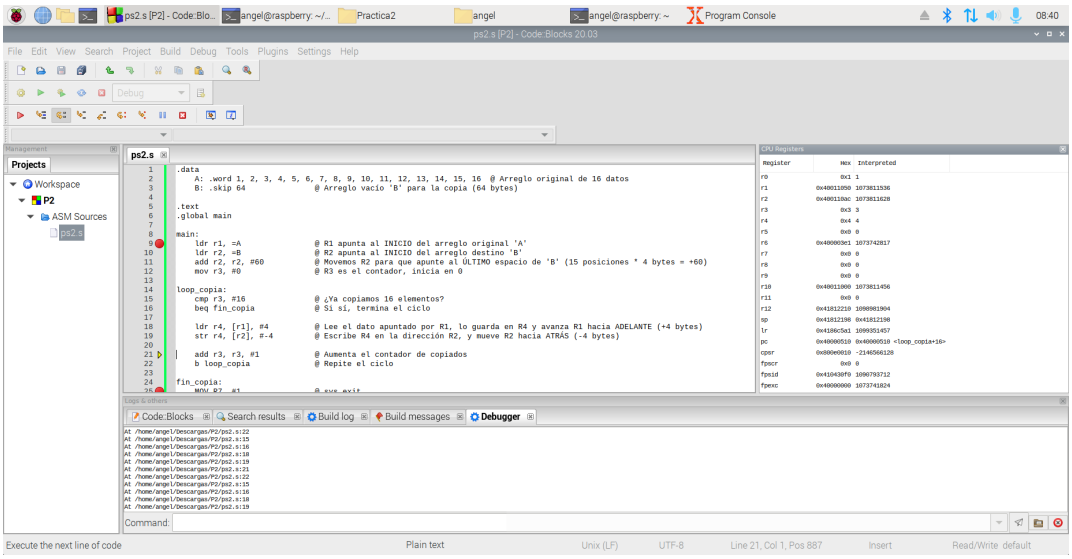


Figura 4: Cuarta iteración. El contador `R3` tiene el valor de 3, y en `R4` se puede observar cargado el valor `0x4`, demostrando que los apuntadores `R1` y `R2` se han actualizado correctamente.

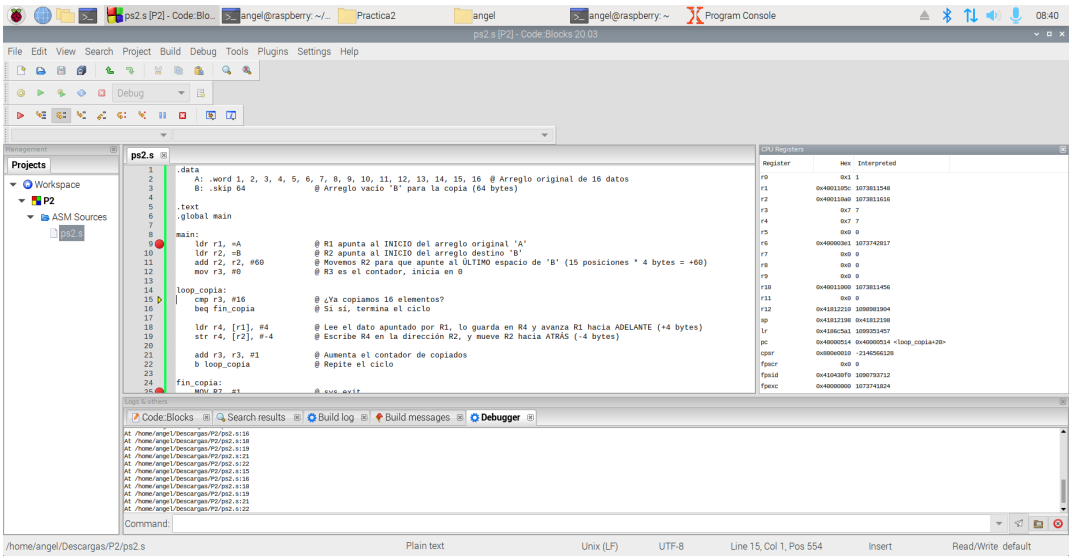


Figura 5: Octava iteración del ciclo. El contador `R3` llega a 7 y en `R4` se carga el valor `0x7`, confirmando la constancia y estabilidad del bucle.

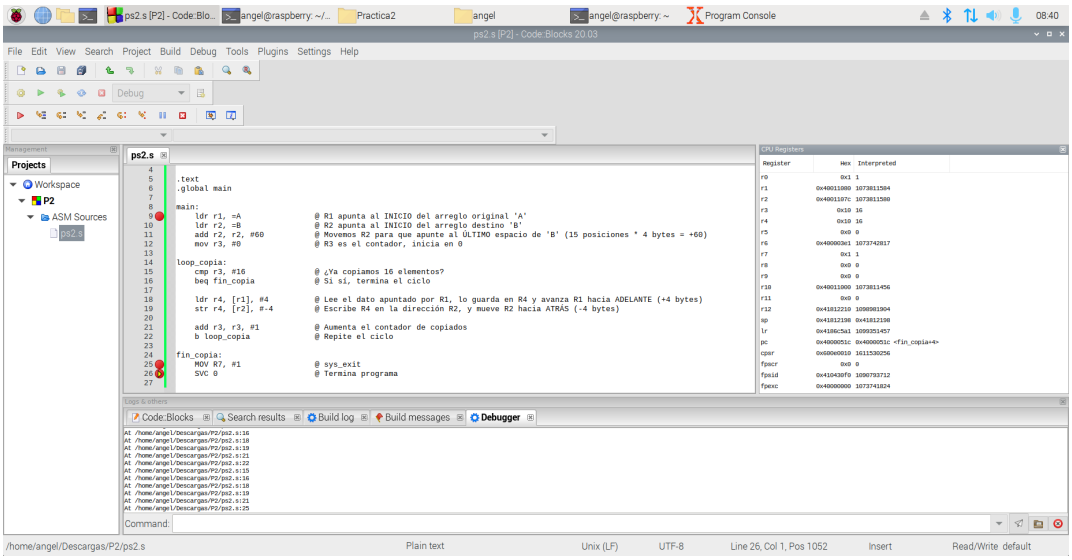


Figura 6: Fin de la ejecución. El contador `R3` alcanza el valor de `0x10` (16 en decimal). El programa sale del ciclo y ejecuta la llamada al sistema (`SVC 0`).

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:

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

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

Desarrollo

Listing 4: Código de la Actividad 4

```
/* ACTIVIDAD 4: Arreglo exponencial y su suma
   Objetivo: Generar serie multiplicando por 2 (Shift), y sumar elementos.
*/
.data
    A:     .skip 80             @ Reserva memoria para 20 elementos (20 * 4 bytes = 80)
    SUMA: .word 0               @ Variable para guardar la sumatoria final

.text
.global main

main:
    ldr r0, =A                  @ R0 apunta a la dirección de memoria de A
```

```
13      mov r1, #3                  @ R1 será la variable 'i' inicial (
            Ejemplo: usamos 3)
14      mov r2, #0                  @ R2 es el contador de elementos
            creados
15      mov r3, #0                  @ R3 será el Acumulador (Sumatoria),
            inicia en 0
16
17  loop_potencias:
18      cmp r2, #20                 @ Compara si ya generamos los 20
            elementos
19      beq fin_potencias           @ Si llegamos a 20, salimos del bucle
20
21      str r1, [r0], #4            @ Guarda el valor actual en memoria y
            avanza el puntero R0
22      add r3, r3, r1              @ Suma el valor actual de 'i' al
            Acumulador Total (R3)
23      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
            SUMA
29      str r3, [r0]                @ Guarda el resultado total (R3) en
            esa memoria
30      MOV R7, #1                  @ sys_exit
31      SVC 0                       @ Termina
```

Análisis de resultados

Actividad 5

Realizar un programa que multiplique dos matrices de 2x2; 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

Desarrollo

Listing 5: Código de la Actividad 5

```
/* ACTIVIDAD 5: Multiplicación de matrices 2x2
   Objetivo: [A B] x [E F] = [I J]
             [C D]   [G H]   [K L]
*/
.data
    M1: .byte 2, 1, 3, 4       @ Matriz 1 (A,B,C,D) -> Valores de
        ejemplo de 8 bits
    M2: .byte 1, 5, 2, 1       @ Matriz 2 (E,F,G,H) -> Valores de
        ejemplo de 8 bits
    MR: .byte 0, 0, 0, 0       @ Matriz Resultado (I,J,K,L)

.text
.global main

main:
    ldr r0, =M1                @ Dirección Matriz 1
    ldr r1, =M2                @ Dirección Matriz 2
    ldr r2, =MR                @ Dirección Matriz Resultado

    @ Cargamos los elementos de M1 (Usamos LDRB por ser Bytes)
    ldrb r3, [r0, #0]          @ R3 = A (Posición 0)
    ldrb r4, [r0, #1]          @ R4 = B (Posición 1)
    ldrb r5, [r0, #2]          @ R5 = C (Posición 2)
    ldrb r6, [r0, #3]          @ R6 = D (Posición 3)

    @ Cargamos los elementos de M2
    ldrb r7, [r1, #0]          @ R7 = E
    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
```

Análisis de resultados

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.

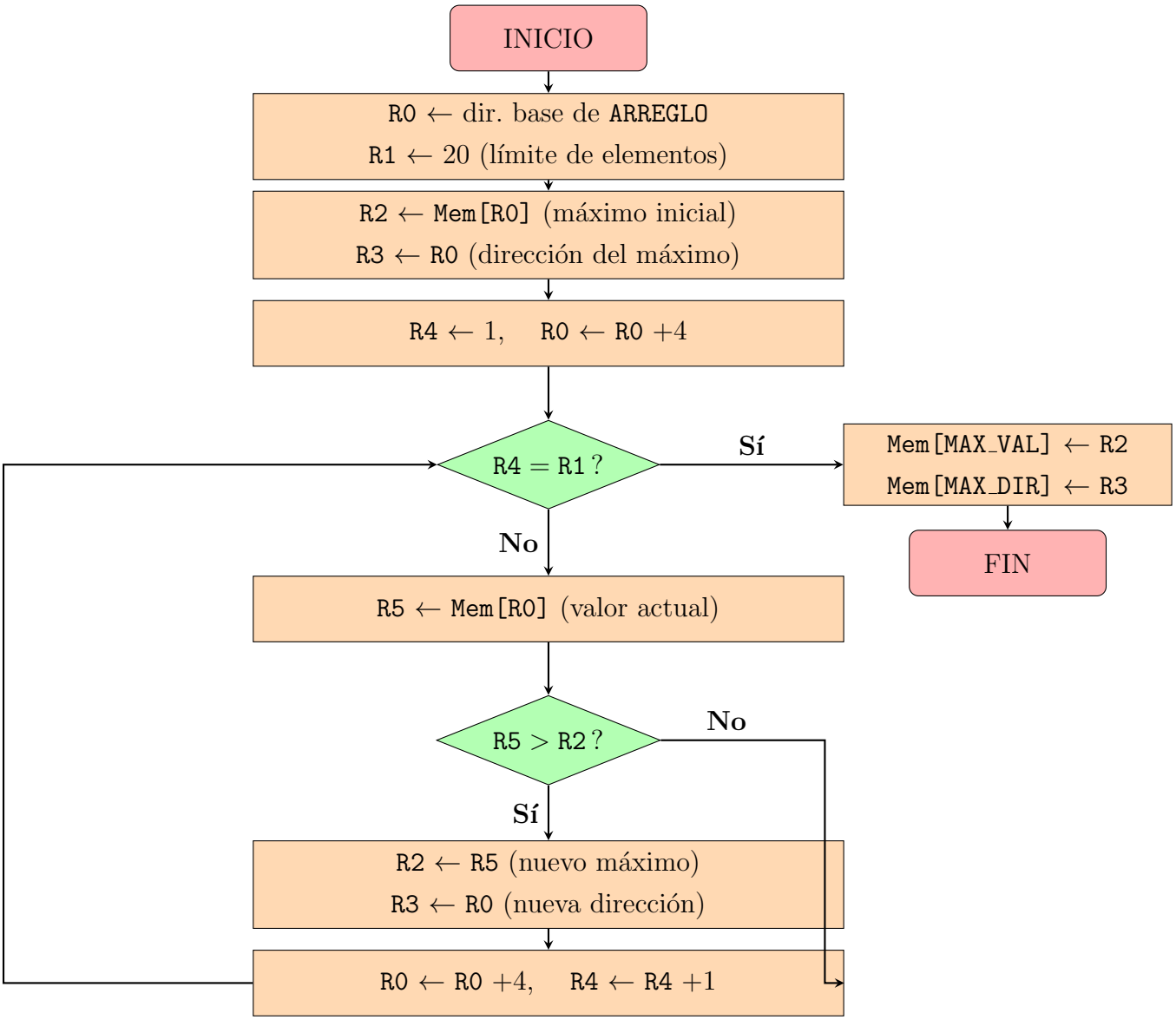
b) Ubicar la dirección donde se encontró este número.

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

```
/* ACTIVIDAD 6: Búsqueda del número mayor en arreglo
   Objetivo: Encontrar el máximo y guardar su valor y su dirección
      de memoria.
*/
.data
    @ Arreglo de 20 números al azar para la prueba
    ARREGLO: .word 5, 12, 3, 45, 2, 105, 1, 8, 33, 10, 11, 14, 0,
       77, 21, 6, 9, 88, 4, 15
    MAX_VAL: .word 0          @ Variable para guardar el número más
       grande
    MAX_DIR: .word 0          @ Variable para guardar la dirección
       de memoria de ese número

.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 = Guarda el MÁXIMO (Inicia
       asumiendo que el índice 0 es el mayor)
    mov r3, r0                @ R3 = Guarda la DIRECCIÓN del máximo
       (Inicia con la del índice 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 í
       ndice 1

```

```
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     ble siguiente               @ Branch if Less or Equal: Si es menor
           o igual, ignóralo y salta a 'siguiente'
28
29     @ Si llegó a esta línea, encontramos un nuevo mayor
30     mov r2, r5                  @ R2 adopta el nuevo valor mayor
31     mov r3, r0                  @ R3 adopta la dirección de memoria de
           este nuevo mayor
32
33 siguiente:
34     add r0, r0, #4              @ Avanzamos la lectura en la memoria
           (4 bytes)
35     add r4, r4, #1              @ Incrementamos el contador de ciclo
36     b buscar_mayor              @ Repetimos
37
38 fin_busqueda:
39     ldr r6, =MAX_VAL            @ Carga dirección para guardar el
           valor
40     str r2, [r6]                @ Almacena en memoria el valor mayor
41     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
```

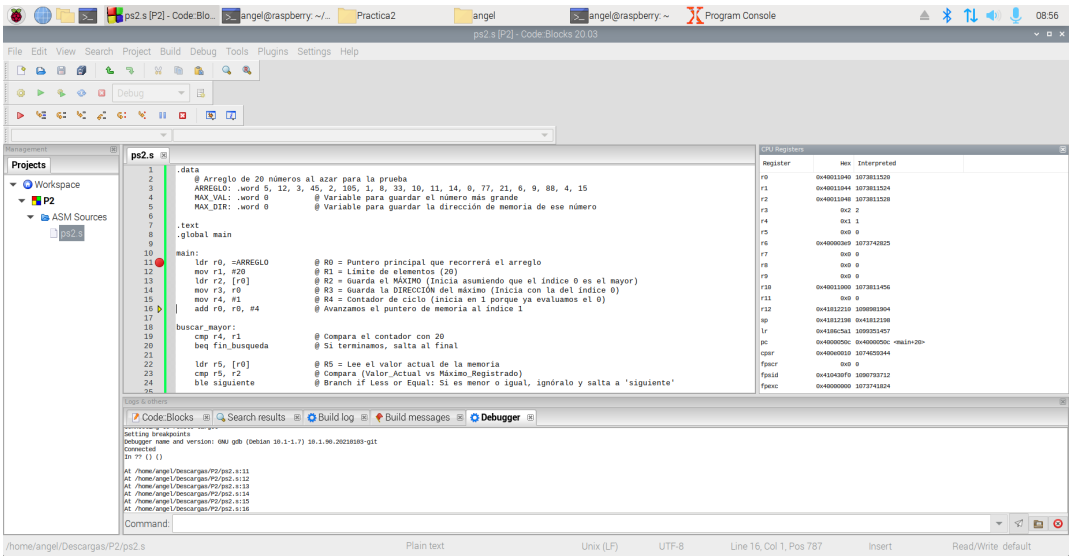


Figura 7: Estado de los registros inmediatamente después de la inicialización de variables. Se asume que el índice 0 es el máximo.

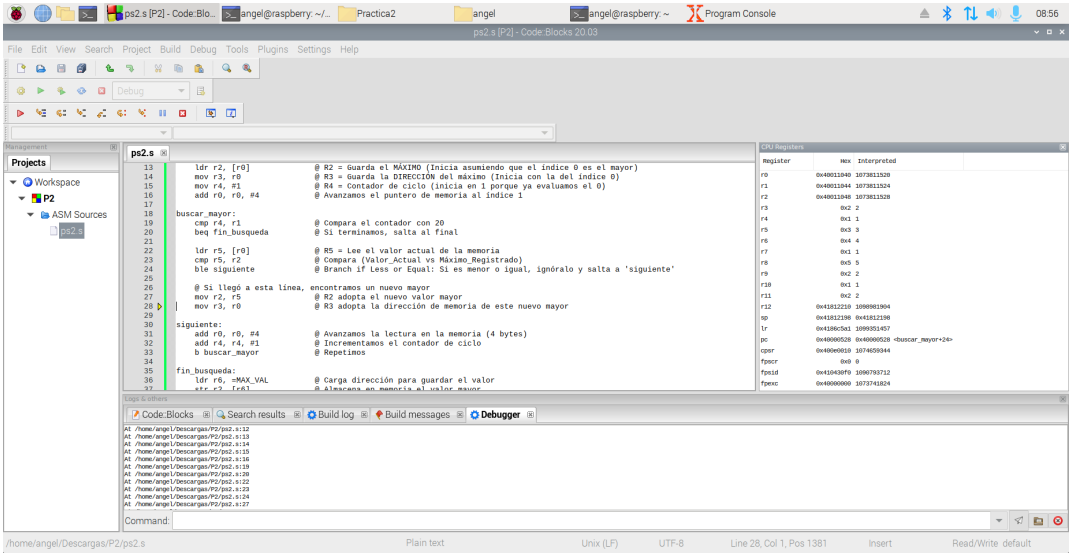


Figura 8: Interrupción dentro del bloque de actualización (`mov r3, r0`) durante la segunda iteración, al encontrar un número mayor que el inicial.

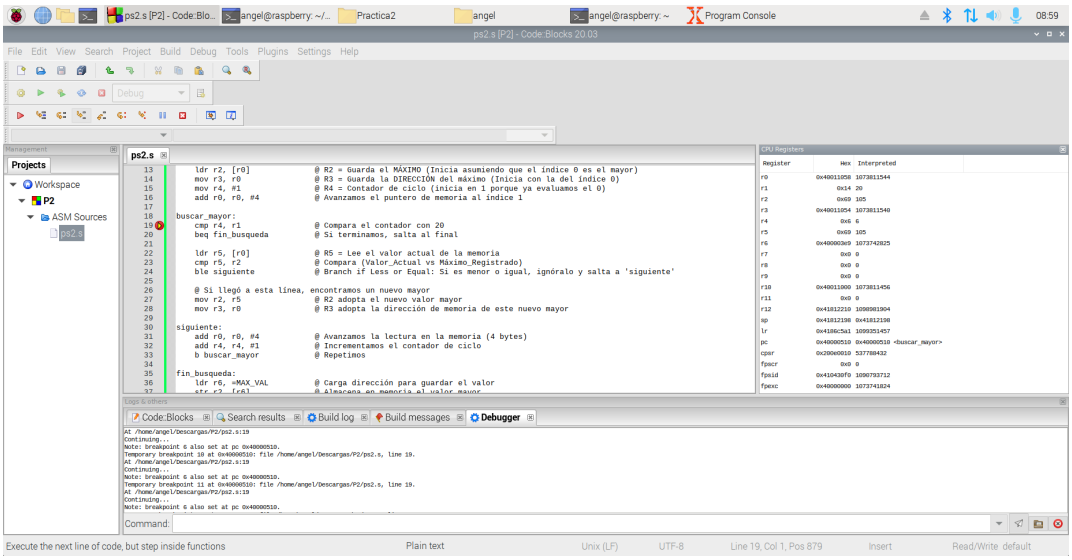


Figura 9: El ciclo cursando la iteración 6 (`R4 = 0x6`). El programa ya ha registrado el verdadero número máximo (`0x69`).

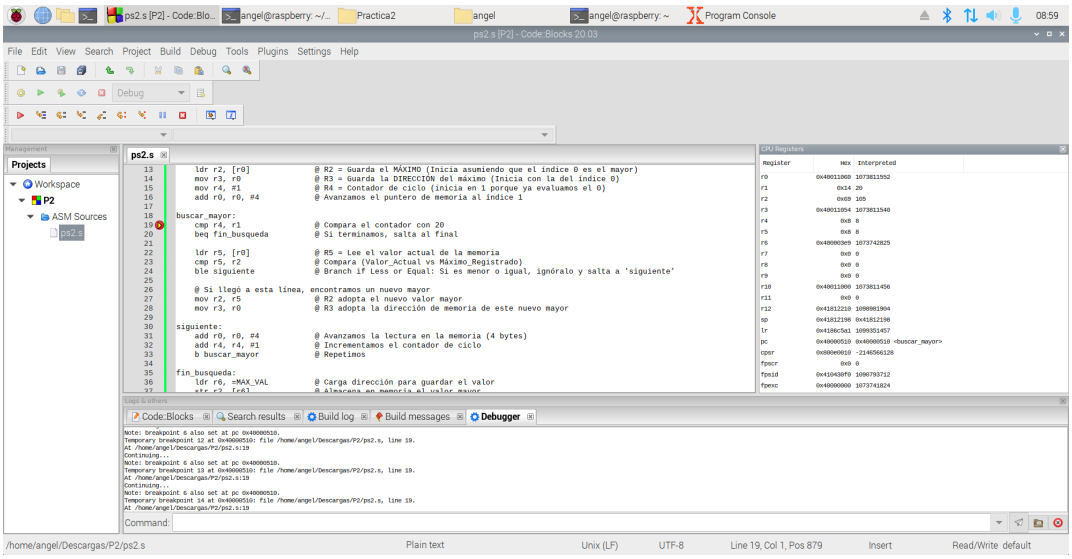


Figura 10: Iteración 8 del ciclo (`R4 = 0x8`). La condición de salto evita que los números menores sobreescriban el valor máximo ya encontrado.

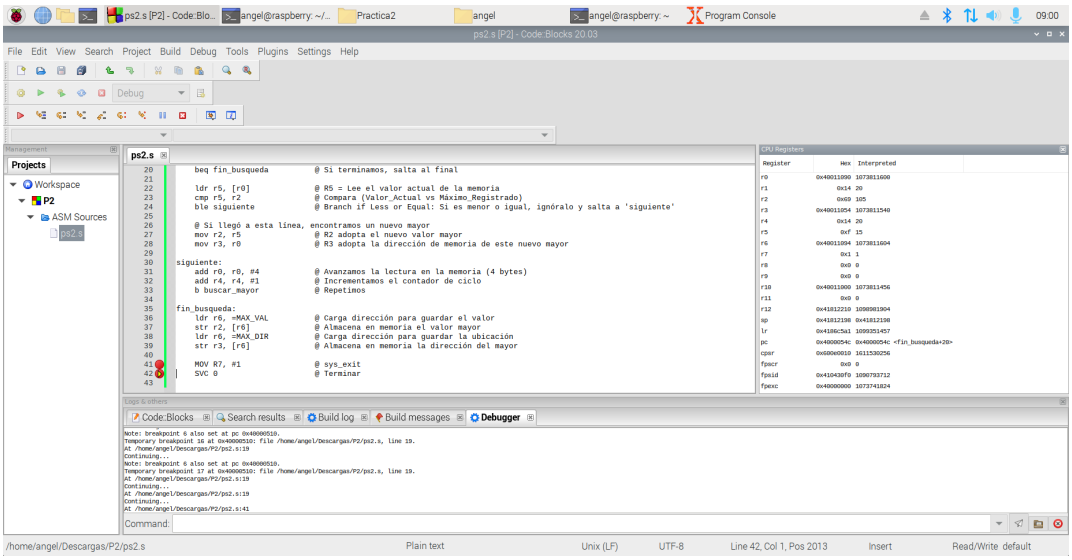


Figura 11: Fin de la ejecución (`SVC 0`). El valor máximo y su dirección se han almacenado en memoria correctamente.

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:

a) Mantener el arreglo original.

b) Generar otro arreglo con el ordenamiento del original.

Arreglo original.

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

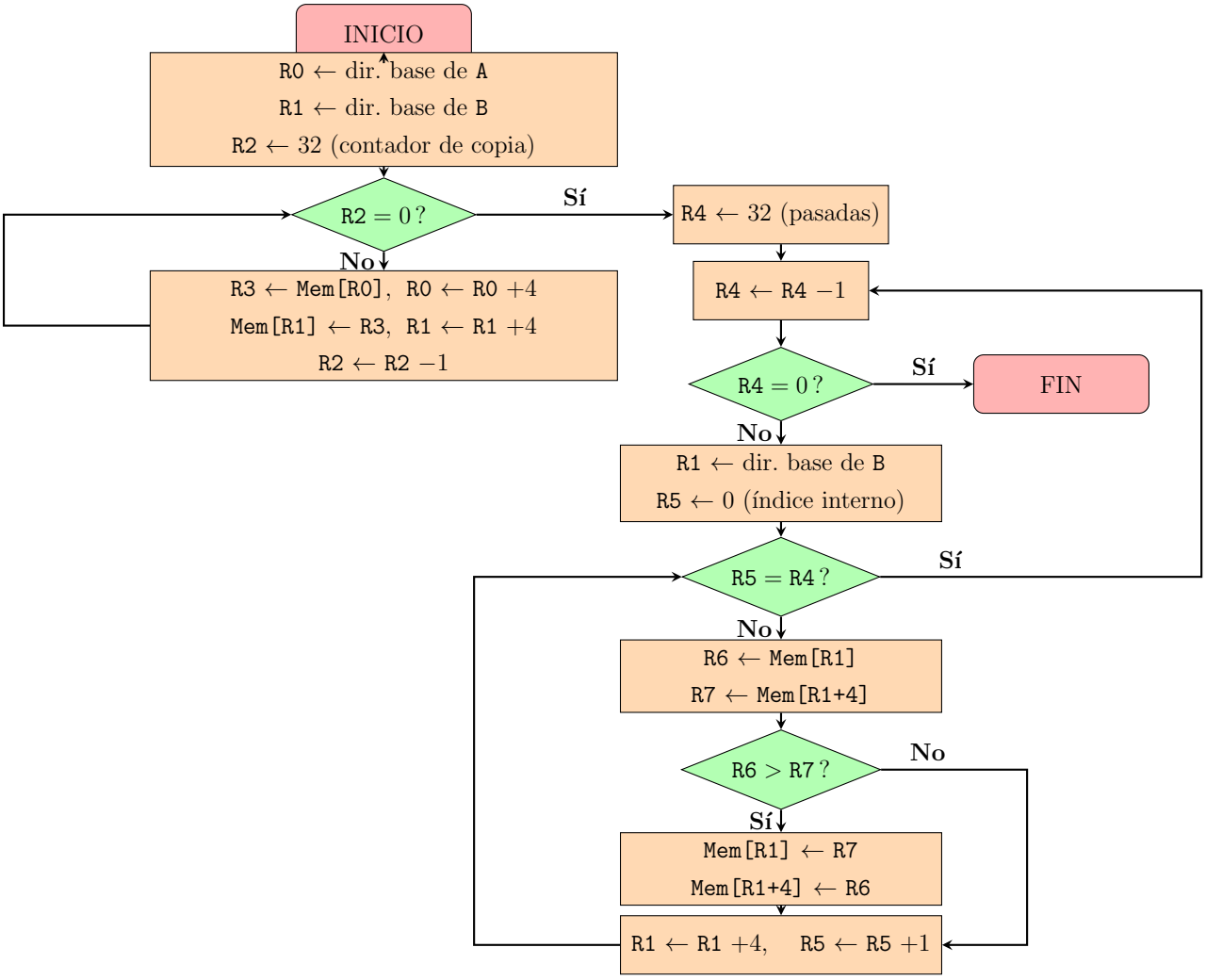
Arreglo ordenado.

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 decrementa el límite de comparaciones en cada pasada (de 32 hasta 0), mientras que el bucle interno recorre los elementos adyacentes de `B` comparándolos entre sí; cuando el elemento izquierdo es mayor que el derecho, se efectúa un intercambio cruzado mediante instrucciones `STR` que escriben los valores en posiciones invertidas. Este proceso se repite hasta que el bucle externo agota sus pasadas, dejando el arreglo `B` completamente ordenado de menor a mayor.

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



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

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

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

Desarrollo

Listing 7: Código de la Actividad 7

```
/* ACTIVIDAD 7: Ordenamiento Burbuja de 32 elementos (32 bits)
   Objetivo: Conservar arreglo original, ordenar la copia.
*/
.data
    @ Arreglo original desordenado (32 elementos)
    A: .word
       32,31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,
    12,11,10,9,8,7,6,5,4,3,2,1
    @ Arreglo copia donde se hará el ordenamiento
    B: .skip 128              @ Reserva 128 bytes (32 words x 4)

.text
.global main

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:
    cmp r2, #0                @ ¿Quedan elementos por copiar?
    beq iniciar_orden         @ Si es 0, terminamos de copiar y
       vamos a ordenar
    ldr r3, [r0], #4          @ Lee de A y avanza
    str r3, [r1], #4          @ Escribe en B y avanza
    sub r2, r2, #1            @ Resta 1 al contador
    b copiar
```

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

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

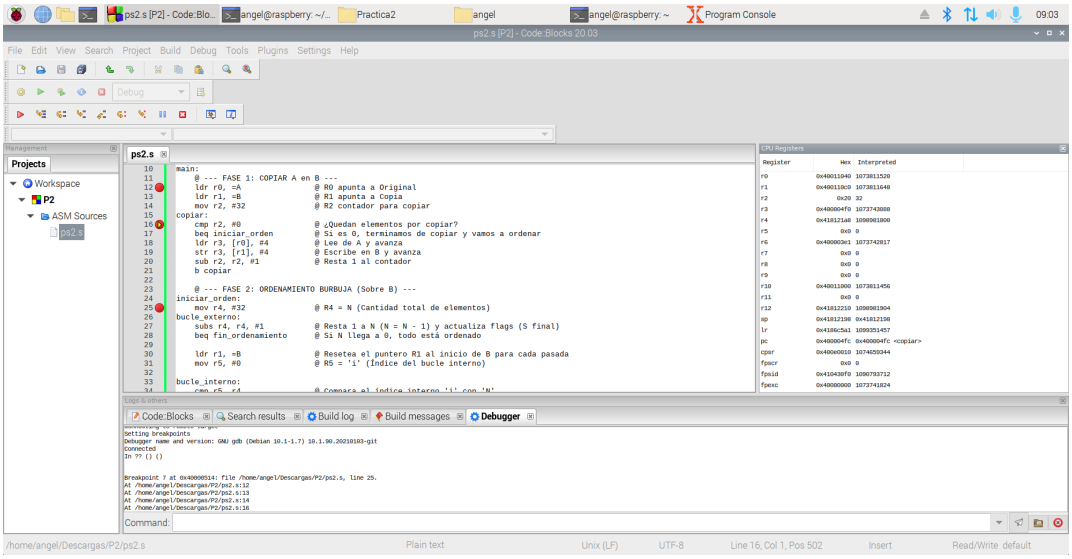


Figura 12: Fase 1 (Copiado). Inicialización de los apuntadores base para el arreglo original (R0) y el arreglo copia (R1).

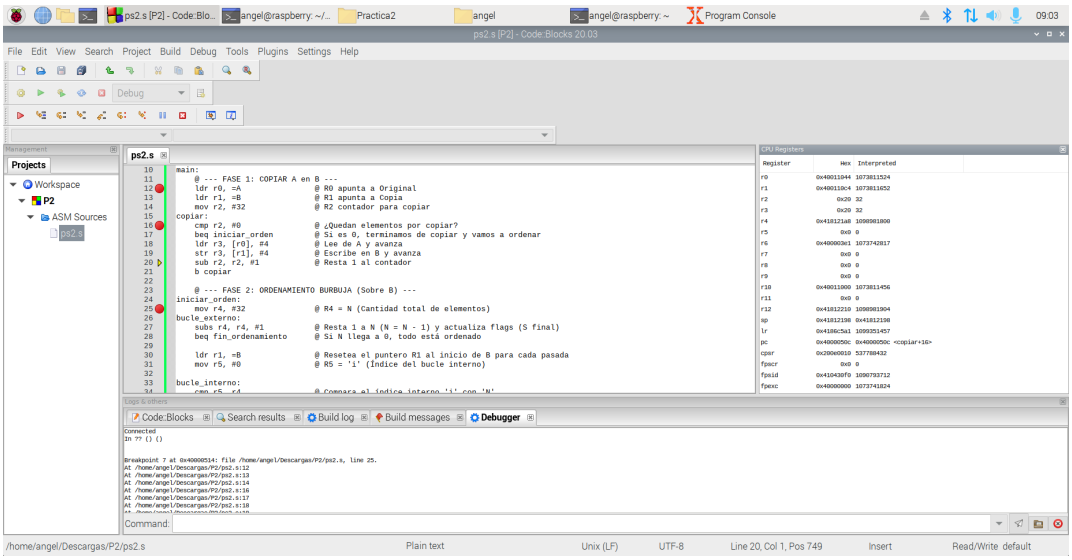


Figura 13: Primera iteración del ciclo de copiado. El registro `R3` extrae el primer valor a transferir.

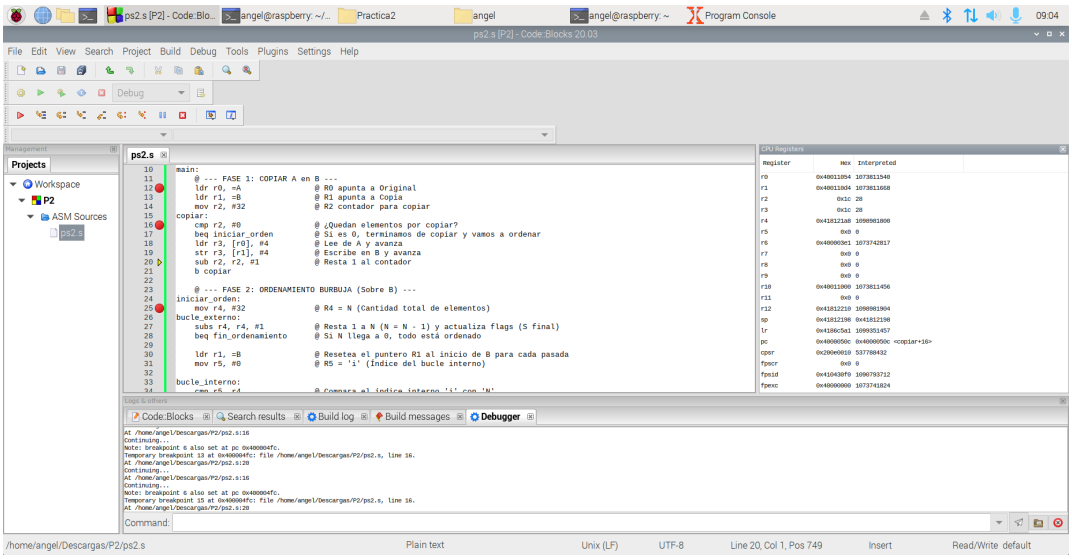


Figura 14: Avance del ciclo de copiado en la iteración correspondiente al quinto elemento transferido (`R2 = 0x1C`).

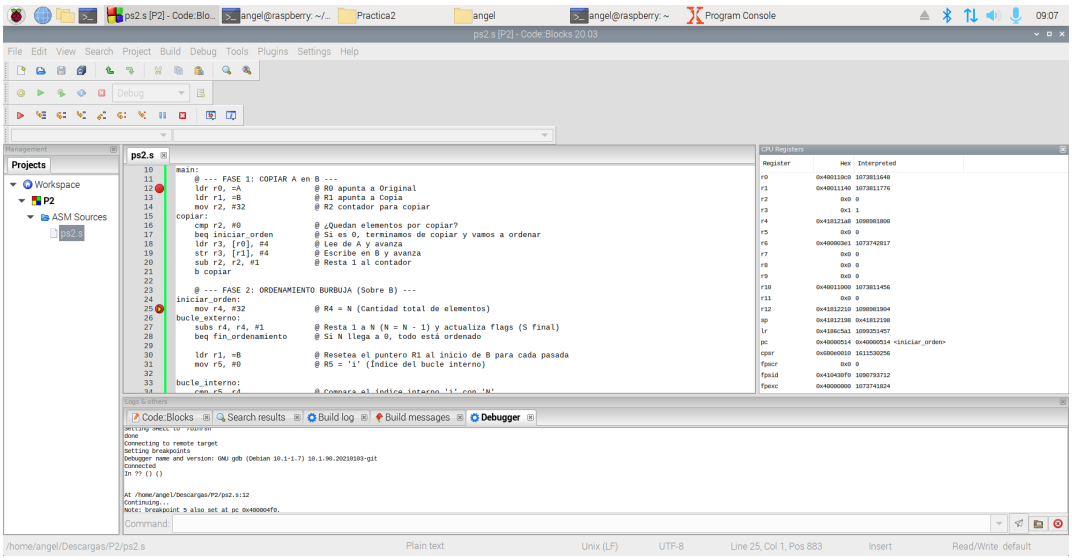


Figura 15: Fin de la fase 1 e inicio de la fase 2. Los apuntadores han recorrido 128 bytes en memoria.

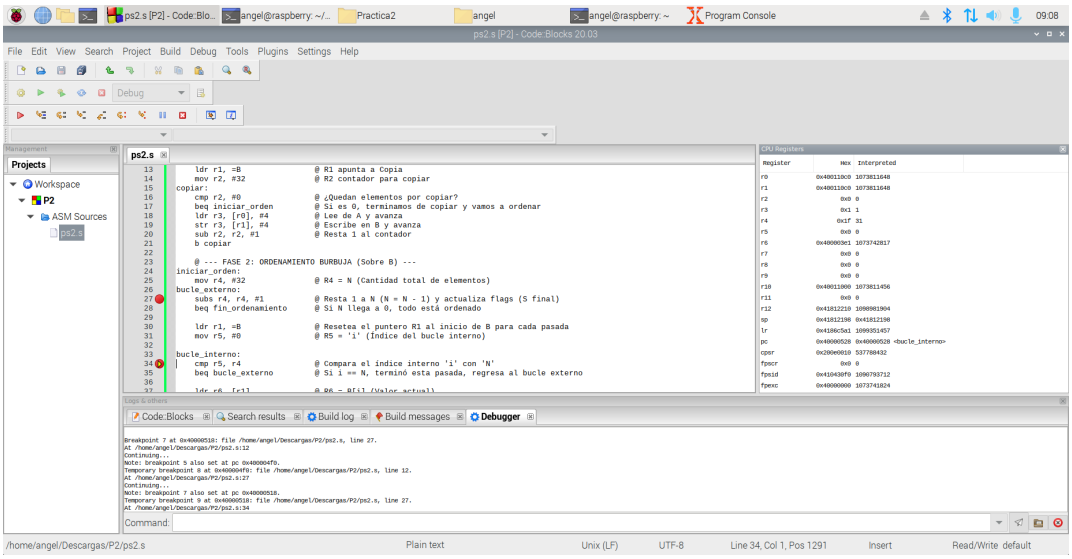


Figura 16: Inicialización del bucle externo del algoritmo de burbuja. `R1` se reinicia a la dirección base de la copia.

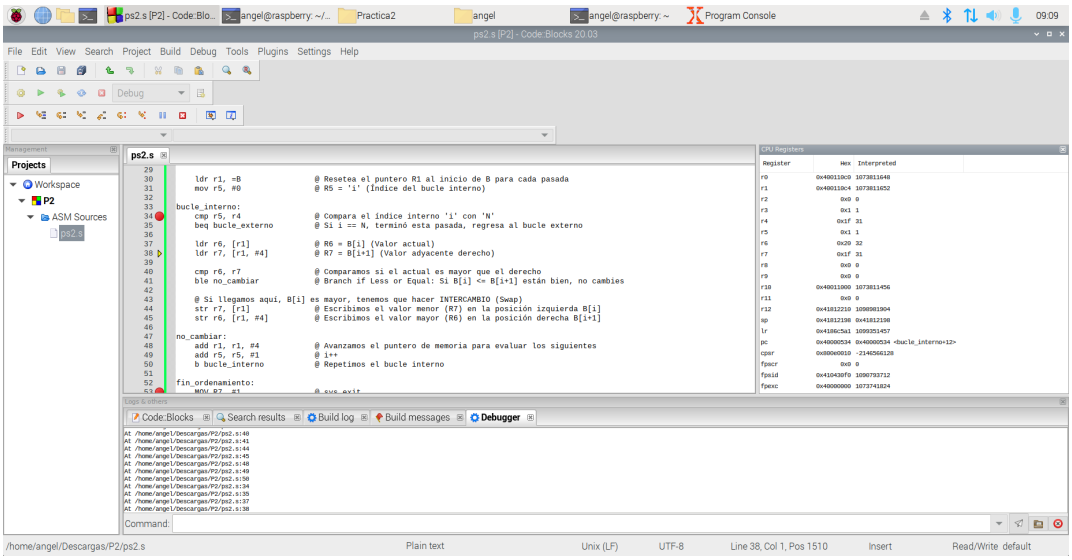


Figura 17: Carga de elementos adyacentes en `R6` y `R7` para su respectiva comparación.

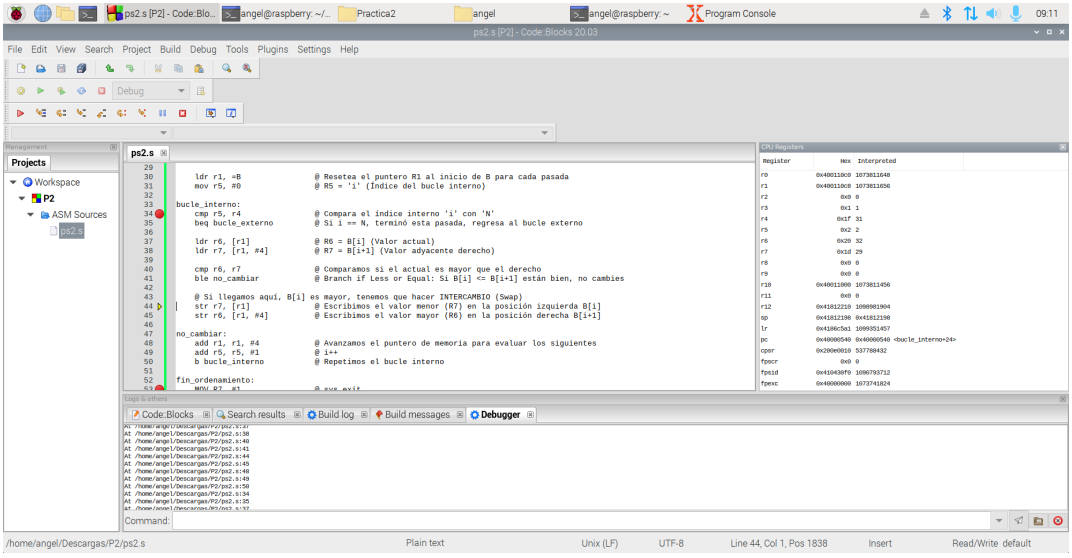


Figura 18: Ejecución del intercambio (*swap*) al detectar que el elemento izquierdo es mayor que el derecho.

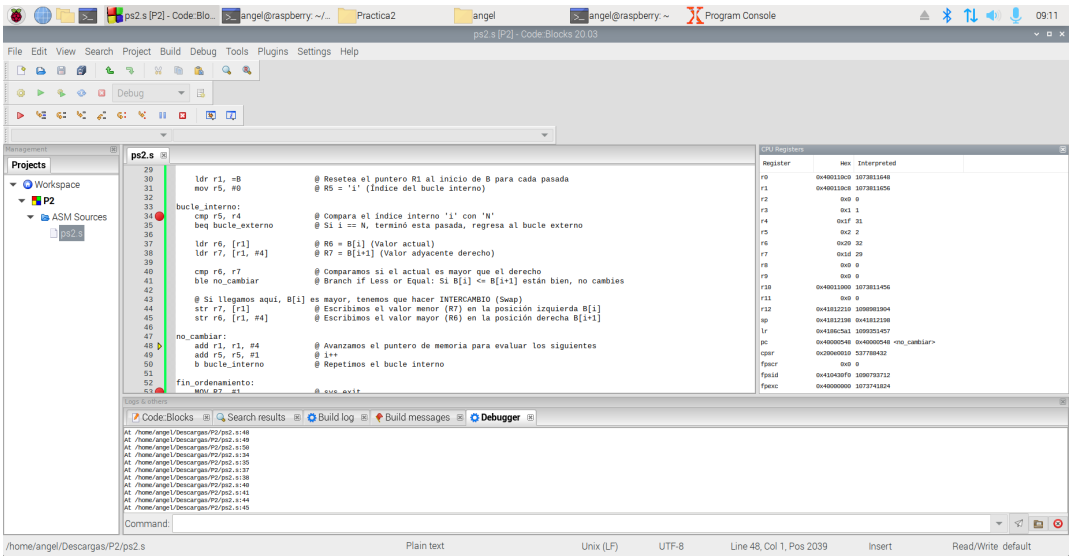


Figura 19: Actualización del apuntador interno `R1` y del contador interno `R5` antes de la siguiente comparación.

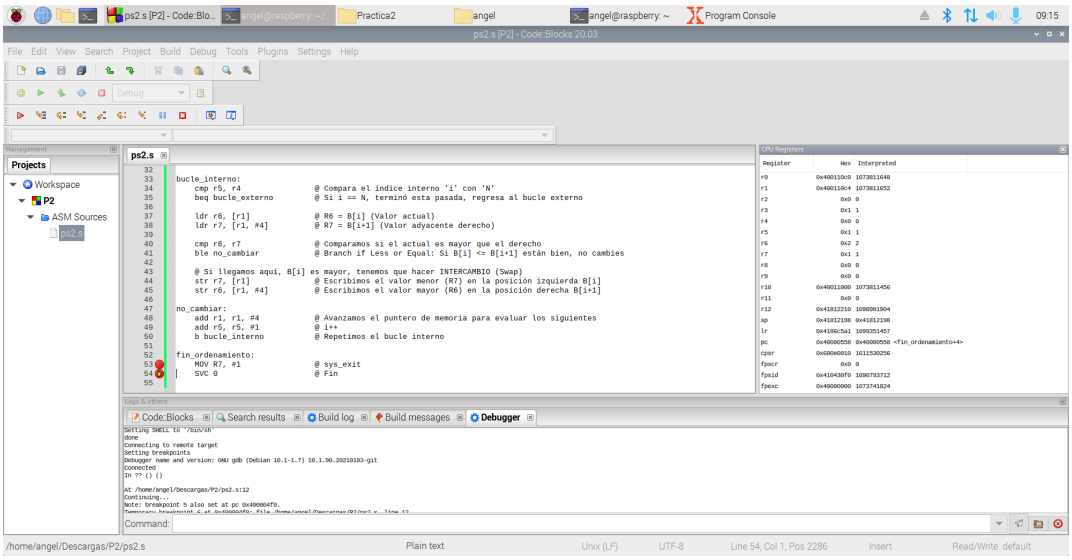


Figura 20: Finalización del programa. Los contadores de ambos bucles agotaron sus iteraciones.

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 reinicializa 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:**
- **Flores Colin Victor Jaziel:**
- **Lara Hernandez Angel Husiel:**

Referencias

Anaya, R. (s.f.). *Manual de recursos y aplicaciones Plataforma Rasberry 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