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

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
          @ Arreglo original de 16 datos
6      B: .skip 64                 @ Arreglo vacío 'B' para la copia (64
          bytes)
7
8  .text
9  .global main
10
11 main:
```

```
12      ldr r1, =A                  @ R1 apunta al INICIO del arreglo
            original 'A'
13      ldr r2, =B                  @ R2 apunta al INICIO del arreglo
            destino 'B'
14      add r2, r2, #60             @ Movemos R2 para que apunte al ÚLTIMO
             espacio de 'B' (15 posiciones * 4 bytes = +60)
15      mov r3, #0                  @ R3 es el contador, inicia en 0
16
17  loop_copia:
18      cmp r3, #16                 @ ¿Ya copiamos 16 elementos?
19      beq fin_copia               @ Si sí, termina el ciclo
20
21      ldr r4, [r1], #4            @ Lee el dato apuntado por R1, lo
            guarda en R4 y avanza R1 hacia ADELANTE (+4 bytes)
22      str r4, [r2], #-4           @ Escribe R4 en la dirección R2, y
            mueve R2 hacia ATRÁS (-4 bytes)
23
24      add r3, r3, #1              @ Aumenta el contador de copiados
25      b loop_copia                @ Repite el ciclo
26
27  fin_copia:
28      MOV R7, #1                  @ sys_exit
29      SVC 0                       @ Termina programa
```

Análisis de resultados

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
    mov r1, #3                  @ R1 será la variable 'i' inicial (Ejemplo: usamos 3)
    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
    ldrb r9, [r1, #2]          @ R9 = G
    ldrb r10,[r1, #3]          @ R10 = H

    @ 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

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 índice 1

buscar_mayor:
    cmp r4, r1                @ Compara el contador con 20
    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
```

Análisis de resultados

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

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

    @ Arreglo copia donde se hará el ordenamiento
    B: .skip 128              @ Reserva 128 bytes (32 words x 4)

.text
.global main
```

```
12
13 main:
14     @ --- FASE 1: COPIAR A en B ---
15     ldr r0, =A                  @ R0 apunta a Original
16     ldr r1, =B                  @ R1 apunta a Copia
17     mov r2, #32                 @ R2 contador para copiar
18 copiar:
19     cmp r2, #0                  @ ¿Quedan elementos por copiar?
20     beq iniciar_orden           @ Si es 0, terminamos de copiar y
          vamos a ordenar
21     ldr r3, [r0], #4            @ Lee de A y avanza
22     str r3, [r1], #4            @ Escribe en B y avanza
23     sub r2, r2, #1              @ Resta 1 al contador
24     b copiar
25
26     @ --- FASE 2: ORDENAMIENTO BURBUJA (Sobre B) ---
27 iniciar_orden:
28     mov r4, #32                 @ R4 = N (Cantidad total de elementos)
29 bucle_externo:
30     subs r4, r4, #1             @ Resta 1 a N (N = N - 1) y actualiza
          flags (S final)
31     beq fin_ordenamiento        @ Si N llega a 0, todo está ordenado
32
33     ldr r1, =B                  @ Resetea el puntero R1 al inicio de B
           para cada pasada
34     mov r5, #0                  @ R5 = 'i' (Índice del bucle interno)
35
36 bucle_interno:
37     cmp r5, r4                  @ Compara el índice interno 'i' con 'N
          '
38     beq bucle_externo           @ Si i == N, terminó esta pasada,
          regresa al bucle externo
39
40     ldr r6, [r1]                @ R6 = B[i] (Valor actual)
41     ldr r7, [r1, #4]            @ R7 = B[i+1] (Valor adyacente derecho
          )
```

```
42
43      cmp r6, r7                  @ Comparamos si el actual es mayor que
            el derecho
44      ble no_cambiar              @ Branch if Less or Equal: Si B[i] <=
           B[i+1] están bien, no cambies
45
46      @ Si llegamos aquí, B[i] es mayor, tenemos que hacer INTERCAMBIO
            (Swap)
47      str r7, [r1]                @ Escribimos el valor menor (R7) en la
            posición izquierda B[i]
48      str r6, [r1, #4]            @ Escribimos el valor mayor (R6) en la
            posición derecha B[i+1]
49
50  no_cambiar:
51      add r1, r1, #4              @ Avanzamos el puntero de memoria para
            evaluar los siguientes
52      add r5, r5, #1              @ i++
53      b bucle_interno             @ Repetimos el bucle interno
54
55  fin_ordenamiento:
56      MOV R7, #1                  @ sys_exit
57      SVC 0                       @ Fin
```

Análisis de resultados

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