

Código Ensamblador: ARMv4

José Bernardo Barquero Bonilla

Carné: 2023150476

Correo: jos.barquero@estudiantec.cr

Escuela de Ingeniería en Computadores

Instituto Tecnológico de Costa Rica

Resumen—

Palabras clave—

I. INTRODUCCIÓN

En el presente informe se desarrolla una serie de ejercicios y análisis relacionados con la arquitectura ARMv4, empleando programación en lenguaje ensamblador. Este laboratorio tiene como propósito hacer uso de instrucciones de bajo nivel, comprensión del funcionamiento interno del procesador y aplicación de estructuras de control fundamentales en un entorno de simulación.

El uso del ensamblador permite una visión más cercana a la operación del hardware, lo que resulta esencial en áreas como los sistemas embebidos y el diseño digital. A través de esta práctica, se exploran aspectos clave de la arquitectura ARMv4, tales como la manipulación de registros, operaciones aritméticas y lógicas, control de flujo, y acceso directo a direcciones de memoria mapeadas.

Además, se complementa el desarrollo práctico con una sección de investigación teórica orientada a comprender los fundamentos técnicos de esta arquitectura, así como sus particularidades respecto al procesamiento de instrucciones y organización de datos en memoria. La implementación de cada ejercicio es acompañada por simulaciones que validan el funcionamiento correcto del código, así como por su respectiva traducción al lenguaje máquina.

II. MARCO TEÓRICO

A. Tipos de instrucciones en ARMv4

La arquitectura ARMv4 cuenta con un conjunto de instrucciones clasificadas en varios tipos fundamentales, cada uno orientado a una función específica dentro del procesador. Entre los principales tipos de instrucciones se encuentran:

- **Instrucciones de transferencia de datos:** se utilizan para cargar o almacenar información entre registros y memoria. Ejemplos: LDR (Load Register) y STR (Store Register), que permiten mover datos entre memoria y registros [1].
- **Instrucciones aritméticas y lógicas:** permiten realizar operaciones matemáticas y de lógica binaria. Ejemplos: ADD (suma), SUB (resta), AND, ORR, entre otras [2].
- **Instrucciones de comparación:** evalúan condiciones sin almacenar el resultado, como CMP (compare), que afecta los flags del procesador según el resultado [3].

- **Instrucciones de control de flujo:** permiten alterar la secuencia de ejecución del programa, como B (branch), BL (branch with link) y variantes condicionales como BEQ o BNE [4].
- **Instrucciones de manipulación de bits:** útiles en el manejo de hardware embebido, como EOR (XOR) y BIC (bit clear) [2].

B. Set de registros en ARM

La arquitectura ARM cuenta con un conjunto de registros de propósito general y registros especializados que permiten realizar operaciones de procesamiento, control de flujo y gestión del estado del programa. En la arquitectura ARMv4, se definen 16 registros accesibles al programador (R0 a R15), además de registros de estado [1].

- **R0–R12:** Son registros de propósito general. Se utilizan para almacenar valores intermedios, parámetros y resultados de operaciones. No tienen una función fija, aunque por convención algunos pueden ser utilizados como acumuladores o para paso de parámetros [4].
- **R13 (SP - Stack Pointer):** Apunta al tope de la pila. Es fundamental en operaciones de llamada a funciones, interrupciones y almacenamiento temporal [2].
- **R14 (LR - Link Register):** Almacena la dirección de retorno cuando se realiza una llamada a subrutina con la instrucción BL. Permite regresar al punto de llamada original una vez finaliza la función [1].
- **R15 (PC - Program Counter):** Contiene la dirección de la próxima instrucción a ejecutar. Su valor se incrementa automáticamente a medida que se ejecutan instrucciones, y puede ser modificado para alterar el flujo del programa [3].
- **CPSR (Current Program Status Register):** Contiene flags que reflejan el estado del procesador (como cero, negativo, acarreo, desbordamiento), así como el modo de operación y el estado de interrupciones. Estos flags son cruciales para decisiones condicionales en el código ensamblador [2].

C. Funcionamiento del branch

En la arquitectura ARMv4, las instrucciones de *branch* permiten modificar el flujo de ejecución de un programa. Estas instrucciones son fundamentales para implementar estructuras de control como bucles, condicionales, llamadas a subrutinas y manejo de excepciones [2].

La instrucción básica es B (branch), que realiza un salto incondicional a una etiqueta o dirección específica. Su funcionamiento se basa en actualizar el contenido del registro PC (R15) con una nueva dirección, permitiendo así que la próxima instrucción ejecutada sea distinta a la secuencia lineal normal [1].

Además del salto incondicional, ARMv4 permite realizar saltos condicionales mediante sufijos que dependen del estado de los *flags* del registro CPSR. Por ejemplo:

- BEQ: salto si el resultado previo fue igual a cero (flag Z = 1).
- BNE: salto si el resultado fue distinto (flag Z = 0).
- BGT, BLT, BGE, BLE: permiten comparaciones con signo.

La instrucción BL (branch with link) permite realizar llamadas a subrutinas, almacenando la dirección de retorno en el registro LR (R14), lo que permite luego regresar con una instrucción como MOV PC, LR [3].

El conjunto de instrucciones *branch* en ARMv4, al ser ensambladas, codifican un desplazamiento relativo a la dirección actual del programa, lo que las hace eficientes y compactas para dispositivos con restricciones de memoria [4].

D. Implementación de condicionales (if / else) en ARMv4

En ARMv4, las estructuras condicionales como if / else no se implementan mediante instrucciones dedicadas de alto nivel, sino a través de instrucciones de comparación y saltos condicionales, aprovechando el conjunto de flags del procesador contenidos en el registro CPSR [5].

El flujo básico para implementar una condición en ensamblador ARMv4 es el siguiente:

- 1) Se realiza una comparación mediante la instrucción CMP, que evalúa dos registros y actualiza los flags del procesador (Z, N, C, V).
- 2) Se emplean instrucciones de salto condicional como BEQ (Branch if Equal), BNE (Branch if Not Equal), BGT (Branch if Greater Than), BLT (Branch if Less Than), entre otras, que toman decisiones basadas en los flags modificados por CMP [6].
- 3) Si se requiere una rama alternativa (else), se utiliza un salto incondicional (B) para evitar ejecutar código no deseado después del bloque condicional verdadero.

Por ejemplo, una estructura if (R0 == R1) se traduce en ensamblador ARMv4 como:

```
CMP R0, R1
BNE else_label
; código para if verdadero
B end_if
else_label:
; código para else
end_if:
```

Esta implementación refleja el modelo de arquitectura RISC de ARM, el cual se basa en instrucciones simples y explícitas, sin estructuras complejas como ocurre en lenguajes de alto

nivel [7]. A pesar de ello, permite construir cualquier tipo de lógica condicional de forma modular y eficiente.

Además, ARM permite usar instrucciones condicionales directamente, como ADDLE o MOVGT, lo que puede simplificar aún más ciertas decisiones sin necesidad de saltos, optimizando el rendimiento en ciclos de reloj [8].

E. Transformación de código ensamblador a binario y herramientas

El proceso de transformación del código ensamblador a lenguaje máquina (binario) es realizado mediante una herramienta conocida como *ensamblador* (assembler). Esta herramienta traduce cada instrucción escrita en lenguaje ensamblador a su representación en código binario, generando así archivos que pueden ser ejecutados directamente por el procesador ARMv4 [9].

Durante este proceso, el ensamblador realiza varias tareas fundamentales:

- Traducción de mnemónicos a instrucciones binarias específicas de la arquitectura.
- Resolución de etiquetas simbólicas utilizadas para saltos y referencias.
- Asignación de direcciones de memoria para datos y código.
- Opcionalmente, puede generar archivos intermedios como listados o archivos de depuración [10].

Para ARMv4, existen múltiples herramientas que permiten realizar esta conversión. Algunas de las más comunes incluyen:

- **GNU Assembler (GAS):** Parte del conjunto GNU Binutils, permite ensamblar código ARM mediante la sintaxis de GNU, generando archivos objeto que luego pueden vincularse y ejecutarse [11].
- **Keil uVision:** Entorno de desarrollo ampliamente utilizado para microcontroladores ARM, que incluye ensamblador y simulador integrado [12].
- **ARM Compiler (armclang):** Parte del entorno de desarrollo oficial de ARM, permite ensamblar y compilar código de forma optimizada para distintos núcleos [13].
- **Herramientas educativas:** Simuladores como VisUAL¹ o CPUlator² permiten escribir, ensamblar y ejecutar código ARMv4 directamente en un entorno web, útiles para fines pedagógicos.

El resultado final del proceso es un archivo binario que representa las instrucciones codificadas en el formato que el procesador puede interpretar directamente. En muchos casos, este binario se almacena en formato hexadecimal o en archivos ejecutables como ELF (*Executable and Linkable Format*) [14].

F. Diferencia entre little endian y big endian

Los términos *little endian* y *big endian* describen dos formas distintas de almacenar los bytes que componen una palabra en la memoria de un sistema. Estas convenciones afectan

¹<https://salmanarif.bitbucket.io/visual/>

²<https://cpulculator.01xz.net/?sys=arm>

directamente cómo se interpretan y transfieren datos entre memoria, registros y dispositivos externos, especialmente en arquitecturas como ARM, que pueden operar en ambos modos dependiendo de la configuración [15].

- En el formato **little endian**, el byte menos significativo se almacena en la dirección de memoria más baja, y los bytes más significativos se almacenan en direcciones sucesivas más altas. Es el formato predeterminado en la mayoría de procesadores ARM actuales [16].
- En el formato **big endian**, el byte más significativo se ubica en la dirección de memoria más baja, y el menos significativo en la más alta. Este enfoque se utilizaba históricamente en algunas arquitecturas como Motorola 68k y SPARC [17].

Por ejemplo, la palabra hexadecimal $0x12345678$ se almacena en memoria como:

- **Little endian:** 78 56 34 12
- **Big endian:** 12 34 56 78

III. DESARROLLO

En la presente sección se comenta el desarrollo realizado para cada uno de los ejercicios planteados en el laboratorio con su respectivo diagrama de flujo.

A. Ejercicio 1

Para el primer ejercicio, se implementó un programa en ensamblador ARMv4 que recorre un arreglo de diez elementos almacenado en memoria, aplicando una operación condicional a cada uno de sus valores. Esta operación consiste en multiplicar el elemento por una constante y si su valor es mayor o igual a dicha constante, o sumarle y si es menor. El programa fue desarrollado con compatibilidad para el emulador VisUAL, por lo que se adoptaron convenciones específicas en la escritura del código, como el uso de mayúsculas en las instrucciones, etiquetas sin dos puntos, alineación con tabulaciones y uso explícito de direcciones relativas.

El diseño se estructuró en tres bloques principales:

- **Inicialización del arreglo:** Se asignó a cada posición del arreglo un valor entero del 0 al 9. El arreglo se almacenó a partir de la dirección base $0x0100$, y se utilizó un bucle `InitLoop` con desplazamientos `LSL #2` para calcular las posiciones correspondientes.
- **Procesamiento condicional:** Utilizando un segundo bucle, se recorrió cada elemento y se comparó con la constante y (igual a 5 en este caso). Se usaron instrucciones de salto condicional como `BLT` y bloques diferenciados (`ThenBranch` y `ElseBranch`) para realizar las operaciones aritméticas necesarias. Las instrucciones `MUL` y `ADD` fueron utilizadas para modificar los valores directamente en memoria.
- **Visualización del resultado:** Con el fin de verificar los cambios en VisUAL, los resultados procesados se cargaron desde memoria a registros visibles (`R5` a `R14`).

El diagrama de flujo correspondiente a la lógica de este ejercicio se muestra en la Figura 1.

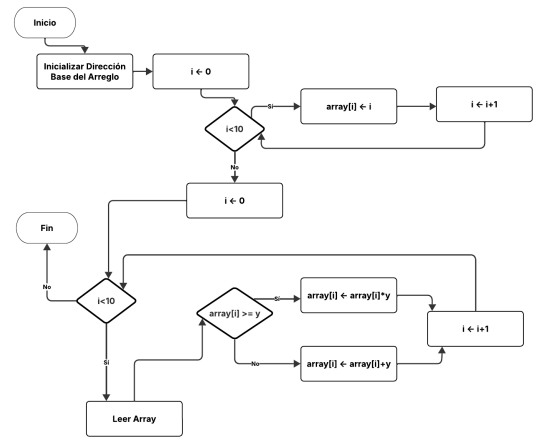


Fig. 1. Diagrama de flujo para el ejercicio 1

B. Ejercicio 2

El segundo ejercicio consistió en implementar un algoritmo para el cálculo del factorial de un número natural X utilizando ensamblador ARMv4. El valor del número se almacenó directamente en un registro para facilitar las pruebas (por ejemplo, $X = 5$), y el resultado se calculó utilizando un enfoque iterativo. El diseño se optimizó para ser compatible con el emulador VisUAL, respetando su sintaxis y convenciones.

El programa se dividió en los siguientes bloques funcionales:

- **Inicialización:** Se definió el valor X en el registro `R0`, el acumulador en `R1` (inicializado en 1), y un contador de iteración `R2` con valor inicial 1. Estos registros permitieron un control claro del proceso sin interferencia con otros valores.
- **Bucle de multiplicación:** Se utilizó un bucle `Loop` controlado mediante la comparación entre el contador `R2` y el valor de entrada `R0`. En cada iteración, el acumulador `R1` se multiplicaba por el contador, utilizando la instrucción `MUL`. El contador se incrementaba con `ADD` hasta alcanzar el valor de X .
- **Resultado final:** Una vez completado el ciclo, el valor final del factorial se almacenaba en `R1`, y se copiaba a `R3` para facilitar su visualización en el entorno del emulador. Esta separación permite mantener intacto el acumulador por si se requiere reutilización posterior.

El diagrama de flujo que representa el diseño lógico de este ejercicio puede verse en la Figura 2.

ensamblador. En la Figura 5 se muestra ambos comandos ejecutados, lo que evidencia el paso final de compilación del código fuente hacia instrucciones directamente interpretables por un procesador ARM.

```

joseph@DESKTOP-03JVSJ3:~/Lab4-CE3201/Ejercicio 1$ xxd ejercicio1.bin
00000000: 010c a0e3 0010 a0e3 0a00 51e3 0300 000a .....Q.....
00000010: 0120 a0e1 0121 a0e7 0110 81e2 f9ff ffea .....!.....
00000020: 0560 a0e3 0010 a0e3 0a00 51e3 0a00 000a .....Q.....
00000030: 0121 a0e1 0230 90e7 0e00 53e1 0200 000a .....0.S.....
00000040: 9306 04e0 0240 80e7 0100 000a 0540 83e0 .....@.....
00000050: 0240 80e7 0110 81e2 f2ff ffea 0050 90e5 .....@.....P..
00000060: 0460 90e5 0870 90e5 0c80 90e5 1090 90e5 .....@.....P..
00000070: 14a0 90e5 18b0 90e5 1c00 90e5 20d0 90e5 .....@.....P..
00000080: 24e0 90e5 .....$...
joseph@DESKTOP-03JVSJ3:~/Lab4-CE3201/Ejercicio 1$ hexdump -C ejercicio1.bin
00000000 01 0c a0 e3 00 10 a0 e3 0a 00 51 e3 03 00 00 0a |.....Q.....|
00000010 01 20 a0 e1 01 21 a0 e7 01 10 81 e2 f9 ff ff ea |.....!.....|
00000020 05 60 a0 e3 00 10 a0 e3 0a 00 51 e3 0a 00 00 0a |.....Q.....|
00000030 01 21 a0 e1 02 30 90 e7 0e 00 53 e1 02 00 00 0a |.....0.S.....|
00000040 93 06 04 e0 02 40 80 e7 01 00 00 0a 05 40 83 e0 |.....@.....|
00000050 02 40 80 e7 01 10 81 e2 f2 ff ff ea 00 50 90 e5 |.....@.....P..|
00000060 04 60 90 e5 08 70 90 e5 0c 80 90 e5 10 90 90 e5 |.....@.....P..|
00000070 14 a0 90 e5 18 b0 90 e5 1c 00 90 e5 20 d0 90 e5 |.....@.....P..|
00000080 24 e0 90 e5 .....$...
00000084

```

Fig. 5. Comandos para ver el código máquina del ejercicio 1

B. Ejercicio 2

En el segundo ejercicio se implementó un algoritmo iterativo para calcular el factorial de un número natural. Para validar el correcto funcionamiento del programa, se utilizó nuevamente el emulador **CPULator**, ya que permite observar los cambios en los registros a lo largo de la ejecución paso a paso.

El valor del factorial se almacena progresivamente en el registro R1, mientras que el contador R2 se incrementa desde 1 hasta el valor de entrada R0. El producto acumulado final se copia a R3 al terminar el ciclo, por lo que resulta conveniente tomar la captura de pantalla justo al finalizar la última iteración, una vez que el salto condicional al final del bucle dirige la ejecución al final del programa.

En las siguientes imágenes se puede verificar que el valor almacenado en R3 corresponde al factorial del número ingresado, confirmando así que la lógica de multiplicación iterativa fue implementada correctamente.

Factorial para $X = 5$

```

r0 00000005
r1 00000000
r2 00000000
r3 00000000
r4 00000020
r5 00000005
r6 00000006
r7 00000007
r8 00000008
r9 00000009
r10 00000019
r11 0000001e
r12 00000023
sp 00000028
lr 0000002d
pc 0000002d
psr 200001d3

```

Address	Opcode	Disassembly
e3a01001	1	mov r1, #1 ; 0x1
e3a02001	2	mov r2, #1 ; 0x1
Loop:		
e1520000	6	cmp r2, r0
ca000002	7	bgt End_Loop
e0b10291	9	mul r1, r1, r2
e2822001	10	add r2, r2, #1 ; 0x1
0	8	b Loop
ea0fffffa	14	b 0xc (Exc: Loop)
0	14	End_Loop:
e1a03001	16	mov r3, r1

Fig. 6. Registros actualizados para el ejercicio 2 con $X = 5$

Factorial para $X = 3$

```

r0 00000003
r1 00000000
r2 00000004
r3 00000006
r4 00000020
r5 00000005
r6 00000006
r7 00000007
r8 00000008
r9 00000009
r10 00000019
r11 0000001e
r12 00000023
sp 00000028
lr 0000002d
pc 0000002d
psr 200001d3

```

Address	Opcode	Disassembly
e3a01001	1	mov r1, #1 ; 0x1
e3a02001	2	mov r2, #1 ; 0x1
Loop:		
e1520000	6	cmp r2, r0
ca000002	7	bgt End_Loop
e0b10291	9	mul r1, r1, r2
e2822001	10	add r2, r2, #1 ; 0x1
0	8	b Loop
ea0fffffa	14	b 0xc (Exc: Loop)
0	14	End_Loop:
e1a03001	16	mov r3, r1

Fig. 7. Registros actualizados para el ejercicio 2 con $X = 3$

Factorial para $X = 10$

```

r0 0000000a
r1 00375f00
r2 00000000
r3 00000004
r4 0000002d
r5 00000005
r6 00000006
r7 00000007
r8 00000008
r9 00000009
r10 00000019
r11 0000001e
r12 00000023
sp 00000028
lr 0000002d
pc 0000002d
psr 200001d3

```

Address	Opcode	Disassembly
0	1	ugr
e3a0000a	2	mov r0, #10 ; 0xa
e3a01001	2	mov r1, #1 ; 0x1
e3a02001	2	mov r2, #1 ; 0x1
Loop:		
e1520000	6	cmp r2, r0
ca000002	7	bgt End_Loop
e0b10291	9	mul r1, r1, r2
e2822001	10	add r2, r2, #1 ; 0x1
0	8	b Loop
ea0fffffa	14	b 0xc (Exc: Loop)
0	14	End_Loop:
e1a03001	16	mov r3, r1

Fig. 8. Registros actualizados para el ejercicio 2 con $X = 10$

Por otro lado, para observar la representación en lenguaje máquina del código ensamblador, se utilizaron los comandos `xxd` y `hexdump` desde el terminal de **VSCoide** ejecutándose sobre **WSL**. Al igual que en el ejercicio anterior, se ensambló y enlazó el archivo fuente, generando un binario que fue visualizado en formato hexadecimal.

La salida generada por estos comandos puede observarse en la Figura 9. En ella se aprecian las instrucciones codificadas, iniciando desde la dirección `00000000`, lo que permite confirmar que el código fue traducido correctamente a lenguaje máquina ARM.

```

joseph@DESKTOP-03JVSJ3:~/Lab4-CE3201/Ejercicio 2$ xxd ejercicio2.bin
00000000: 0500 a0e3 0110 a0e3 0120 a0e3 0000 52e1 .....R..
00000010: 0200 00ca 9102 01e0 0120 82e2 faff ffea .....@.....
00000020: 0130 a0e1 .....0..
joseph@DESKTOP-03JVSJ3:~/Lab4-CE3201/Ejercicio 2$ hexdump -C ejercicio2.bin
00000000 05 00 a0 e3 01 10 a0 e3 01 20 a0 e3 00 00 52 e1 |.....R..|
00000010 02 00 00 ca 91 02 01 e0 01 20 82 e2 fa ff ff ea |.....@.....|
00000020 01 30 a0 e1 .....0..
00000024

```

Fig. 9. Comandos para ver el código máquina del ejercicio 2

C. Ejercicio 3

El tercer ejercicio consistió en simular el control de la posición de un sprite en pantalla mediante el uso de un contador ubicado en memoria. Este contador debía incrementarse o decrementarse según la tecla leída desde una dirección mapeada, y mantenerse sin cambios en caso de recibir una tecla inválida. La lógica fue implementada utilizando instrucciones condicionales que verifican si la entrada coincide con los valores específicos de las teclas de flecha arriba (`0xE048`) o flecha abajo (`0xE050`).

Para cumplir con los requerimientos del laboratorio, se simuló manualmente el valor de la tecla escribiéndolo directamente en la dirección `0x1000` durante cinco iteraciones del ciclo principal. El orden de las teclas utilizadas fue el siguiente:

- 1) Flecha abajo (`0xE050`) — decremento del contador.
- 2) Flecha arriba (`0xE048`) — incremento del contador.
- 3) Flecha abajo (`0xE050`) — decremento del contador.
- 4) Tecla inválida (`0x1234`) — el contador no se modifica.
- 5) Flecha arriba (`0xE048`) — incremento del contador.

Cada tecla fue escrita justo antes de cerrar la iteración correspondiente del bucle, simulando el cambio de entrada por parte del usuario. Esta secuencia permitió probar los tres casos exigidos por el enunciado: incremento, decremento y mantenimiento del valor.

Para observar el estado final del contador, se utilizó el emulador **CPUlator**, visualizando el contenido del registro R5, que almacena el valor leído desde la dirección 0x2000. La captura mostrada en la Figura 10 corresponde al momento inmediatamente posterior a la quinta iteración, cuando el programa termina su ejecución. En esta imagen puede verificarse que el valor del contador refleja correctamente el resultado esperado después de las operaciones aplicadas.

Register	Value
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000005
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
SP	0x00000000
LR	0x00000000
PC	0x00000020

Fig. 10. Registros actualizados para el ejercicio 3

Al igual que en los ejercicios anteriores, se generó el código de máquina correspondiente utilizando **VSCode** y **WSL**. Se ejecutaron los comandos `xxd ejercicio3.bin` y `hexdump -C ejercicio3.bin` desde la terminal, obteniendo la representación binaria del programa en formato hexadecimal. La Figura 11 muestra una captura de estos comandos, validando que el código fue ensamblado correctamente y está listo para ser ejecutado en un entorno real o simulado.

```

joseph@DESKTOP-03JVS9J:~/Lab4-CE3201/Ejercicio 3$ xxd ejercicio3.bin
00000000: 010a a0e3 021a a0e3 0020 a0e3 0070 81e5 .....J.H.....S.
00000010: 0030 02e5 0e4a a0e3 4040 84e2 0400 53e1 .....J..P.....S.
00000020: 0400 000a 0e4a a0e3 5040 84e2 0400 53e1 .....J..P.....S.
00000030: 0400 000a 0700 000a 0050 91e5 0150 85e2 .....P.....P..P.
00000040: 0050 81e5 0300 000a 0050 91e5 0150 45e2 .....P.....P..P.
00000050: 0050 81e5 ffff ffea cfff ffea .....P.....P..P.
joseph@DESKTOP-03JVS9J:~/Lab4-CE3201/Ejercicio 3$ hexdump -C ejercicio3.bin
00000000 01 0a a0 e3 02 1a a0 e3 00 20 a0 e3 00 70 81 e5 |.....J.H.....S|
00000010 00 30 02 e5 0e 4a a0 e3 40 40 84 e2 04 00 53 e1 |.....J..P.....S|
00000020 04 00 00 0a 0e 4a a0 e3 50 40 84 e2 04 00 53 e1 |.....J..P.....S|
00000030 04 00 00 0a 07 00 00 0a 00 50 91 e5 01 50 85 e2 |.....P.....P..P|
00000040 00 50 81 e5 03 00 00 0a 00 50 91 e5 01 50 45 e2 |.....P.....P..P|
00000050 00 50 81 e5 ff ff ff ea ec ff ff ea |.....P.....P..P|
0000005c

```

Fig. 11. Comandos para ver el código máquina del ejercicio 3

V. CONCLUSIONES

REFERENCIAS

- [1] S. Furber, *ARM System-on-Chip Architecture*. Addison-Wesley, 2000, ISBN: 9780201675191.
- [2] A. N. Sloss, D. Symes y C. Wright, *ARM System Developer's Guide: Designing and Optimizing System Software*. Elsevier, 2004, ISBN: 9781558608740.
- [3] J. Yiu, *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors*. Newnes, 2015, ISBN: 9780128032770.
- [4] P. Lyons, *ARM Assembly Language Programming*. Course Technology, 2004, Manual interno de entrenamiento ARM.
- [5] J. W. Valvano, *Embedded Systems: Introduction to ARM Cortex-M Microcontrollers*. CreateSpace, 2011, ISBN: 9781463590154.

- [6] T. Noergaard, *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Elsevier, 2005, ISBN: 9780750677929.
- [7] D. Harris y S. Harris, *Digital Design and Computer Architecture: ARM Edition*. Morgan Kaufmann, 2021, ISBN: 9780128200643.
- [8] J. Yiu, *The Definitive Guide to ARM Cortex-M0 and Cortex-M0+ Processors*, 3rd. Newnes, 2022, ISBN: 9780323998393.
- [9] J. Lombardi, *Modern Computer Architecture and Organization*. Packt Publishing, 2019, ISBN: 9781838552634.
- [10] F. V. Barrett, *Embedded Systems Design: An Introduction to Processes, Tools, and Techniques*. Prentice Hall, 2006, ISBN: 9780131486930.
- [11] F. S. Foundation, *Using as: The GNU Assembler*, https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_node/as_toc.html, Consultado en mayo de 2025, 2020.
- [12] K. A. Ltd.), *uVision IDE and Debugger*, <https://www.keil.com/uvision/>, Consultado en mayo de 2025.
- [13] A. Ltd., *Arm Compiler 6 – armclang Reference*, <https://developer.arm.com/tools-and-software/embedded/arm-compiler>, Consultado en mayo de 2025.
- [14] J. Ganssle, *The Art of Designing Embedded Systems*. Newnes, 2008, ISBN: 9780750686259.
- [15] D. A. Patterson y J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th. Morgan Kaufmann, 2013, ISBN: 9780124077263.
- [16] Intel Corporation, *Understanding Big and Little Endian Byte Order*, <https://www.intel.com/content/www/us/en/developer/articles/technical/understanding-big-and-little-endian-byte-order.html>, Consultado en mayo de 2025, 2021.
- [17] R. E. Bryant y D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 4th. Pearson, 2022, ISBN: 9780137646321.