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*—En este informe se documenta el desarrollo del Laboratorio 4 del curso CE3201 — Taller de Diseño Digital, cuyo objetivo fue fortalecer el conocimiento sobre la arquitectura ARMv4 mediante la implementación de programas en lenguaje ensamblador. A través de tres ejercicios prácticos, se abordaron conceptos esenciales como el control de flujo, el uso de registros, las operaciones condicionales y la interacción con memoria mapeada. El primer ejercicio consistió en procesar un arreglo de enteros aplicando una operación condicional elemento por elemento. En el segundo, se implementó un algoritmo iterativo para calcular el factorial de un número, reforzando el uso de bucles y multiplicaciones. El tercer ejercicio simuló la interacción con un periférico de entrada (teclado), controlando la posición de un sprite en memoria según las teclas leídas. Se utilizaron los emuladores VisUAL y CPUlator para verificar el comportamiento de los registros y la lógica del programa en tiempo real. Asimismo, se generaron los archivos binarios correspondientes a cada ejercicio utilizando el entorno de desarrollo Visual Studio Code junto con WSL, permitiendo observar el lenguaje máquina mediante los comandos `xxd` y `hexdump`.Los resultados obtenidos validan el funcionamiento correcto de los programas y demuestran la importancia de comprender el vínculo entre código de bajo nivel y ejecución en arquitectura ARM, facilitando el diseño y análisis de sistemas digitales embebidos.**

***Palabras clave*—ARMv4, lenguaje ensamblador, registros, instrucciones condicionales, memoria mapeada, emulador VisUAL, CPUlator, código binario, bucle iterativo, direccionamiento, arquitectura RISC, WSL, VSCode, hexdump.**

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 VisUAL[1] o CPUlator[2] 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 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.

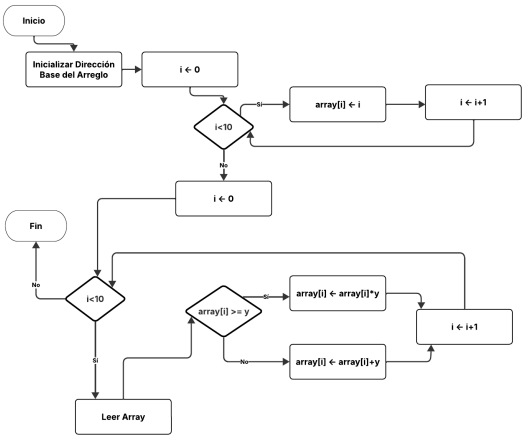

Fig. 1. Diagrama de flujo para el ejercicio 1

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.

[1] https://salmanarif.bitbucket.io/visual/
[2] https://cpulator.01xz.net/?sys=arm

- **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.

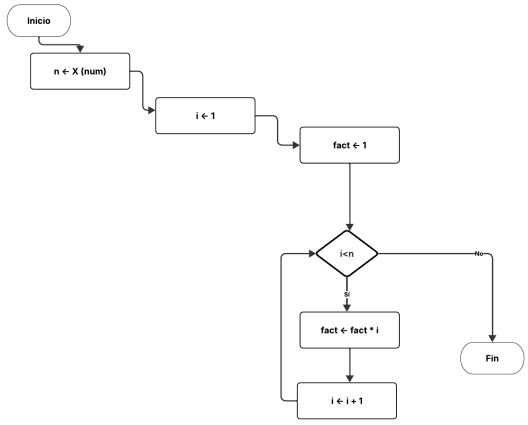

Fig. 2. Diagrama de flujo para el ejercicio 2

C. Ejercicio 3

El tercer ejercicio consistió en simular un entorno donde un sprite representado por un contador en memoria cambia su posición en respuesta a entradas de teclado. El diseño implementado está orientado a sistemas embebidos, donde los periféricos como teclados y pantallas están mapeados en direcciones de memoria específicas. Para este caso, la dirección `0x1000` se utilizó como entrada del teclado, mientras que `0x2000` representó la posición actual del sprite (contador).

El programa se organizó en las siguientes secciones:

- **Inicialización:** Se cargaron en `R0` y `R1` las direcciones de memoria correspondientes al teclado y al contador, respectivamente. Además, se inicializó el contador con el valor 0 mediante una instrucción `STR`.
- **Lectura y evaluación de la entrada:** En un bucle infinito, el programa lee continuamente el valor de la dirección de entrada (`LDR R3, [R0]`). Luego compara dicho valor con las constantes codificadas en hexadecimal para detectar si se ha presionado la flecha hacia arriba (`0xE048`) o hacia abajo (`0xE050`).
- **Procesamiento condicional:** Si la tecla corresponde a la flecha hacia arriba, el programa incrementa el valor en la dirección del contador usando una lectura y suma con `ADD`. Si se detecta la flecha hacia abajo, se realiza una operación similar con `SUB`. Cualquier otro valor es ignorado para asegurar que no ocurran alteraciones indeseadas.
- **Bucle continuo:** Después de procesar o ignorar la entrada, el programa vuelve al inicio del bucle con una instrucción `B`, permitiendo la lectura constante del teclado. Esta estructura es típica en sistemas de control embebidos que operan en tiempo real sin finalización explícita.

Para efectos de prueba en el emulador VisUAL, fue necesario simular la escritura en la dirección de teclado `0x1000` manualmente antes de cada iteración. Se probaron múltiples escenarios, incluyendo entradas válidas (ambas flechas) y al menos una inválida para verificar el correcto comportamiento del sistema.

El diagrama de flujo del programa puede observarse en la Figura 3.

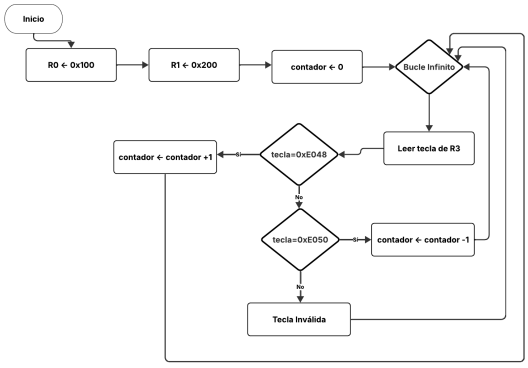

Fig. 3. Diagrama de flujo para el ejercicio 3

IV. Análisis de Resultados

A. Ejercicio 1

Para verificar el funcionamiento del programa del ejercicio 1, se utilizaron dos emuladores: **VisUAL** y **CPUlator**. Ambos permiten observar el contenido de los registros, las instrucciones ejecutadas y los accesos a memoria durante la ejecución del código ensamblador ARMv4. El análisis se centró en validar que los elementos del arreglo fueran correctamente modificados según la condición establecida: multiplicación por la constante y si el valor era mayor o igual, o suma en caso contrario.

Dado que el programa recorre un arreglo en un bucle, se optó por tomar una única captura de pantalla al finalizar la última iteración del ciclo, donde ya se encuentran cargados en los registros `R5` a `R14` los valores finales procesados del arreglo. Esto permite verificar de manera clara y compacta que la lógica condicional se ejecutó correctamente en cada uno de los elementos del arreglo sin saturar el informe con múltiples capturas intermedias. En la Figura 4 se puede apreciar la actualización de los registros en CPUlator.

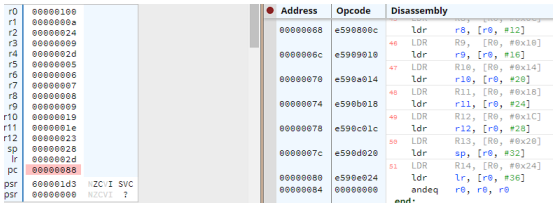

Fig. 4. Registros actualizados para el ejercicio 1

Adicionalmente, se realizó la generación del código de máquina binario utilizando **VSCode** con soporte para **WSL**. Desde el terminal se ejecutaron los siguientes comandos:

```
xxd ejercicio1.bin
hexdump -C ejercicio1.bin
```

Estas herramientas permitieron visualizar la representación hexadecimal del archivo binario generado a partir del código 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.

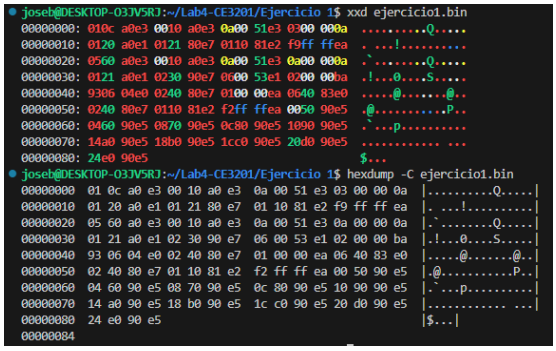

Fig. 5. Comandos para ver el código máquina del ejercicio 1

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 redirige 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 $X = 5$

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

Factorial para $X = 3$

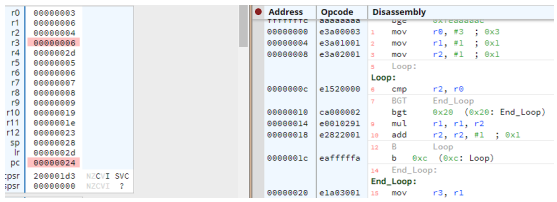

Fig. 7. Registros actualizados para el ejercicio 2 con $X = 3$

Factorial para $X = 10$

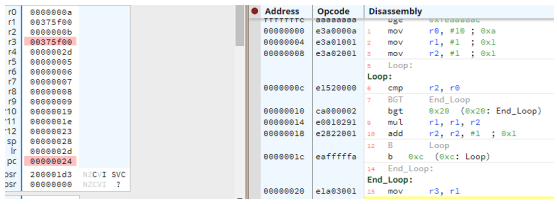

Fig. 8. Registros actualizados para el ejercicio 2 con $X = 10$

Por otro lado, para observar la representación en lenguaje máquina del código ensamblador, se utilizaron los comandos `xxd` y `hexdump` desde el terminal de **VSCode** 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.

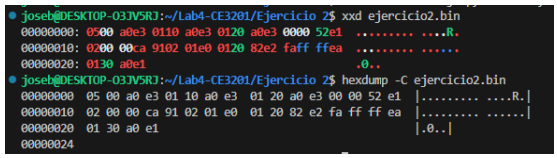

Fig. 9. Comandos para ver el código máquina del ejercicio 2

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.

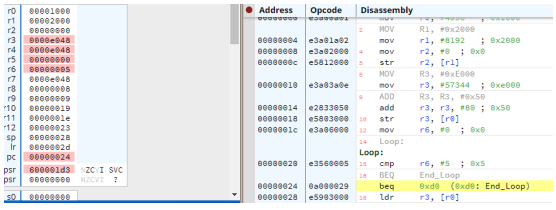

Fig. 10. Registros actualizados para el ejercicio 3

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.

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

V. Conclusiones

El desarrollo de este laboratorio permitió comprender a profundidad el funcionamiento del lenguaje ensamblador en la arquitectura ARMv4, reforzando conocimientos clave sobre control de flujo, manipulación de registros y operaciones condicionales.

Se logró no solo codificar instrucciones efectivas para resolver problemas clásicos, sino también validar su ejecución utilizando emuladores como VisUAL y CPUlator, que brindaron retroalimentación visual y precisa.

La generación de archivos binarios desde el entorno WSL en VSCode facilitó el entendimiento del proceso completo de transformación del código fuente en lenguaje máquina, fortaleciendo la conexión entre software y hardware.

La prueba manual de entradas y la visualización de resultados demostraron que un enfoque meticuloso en el diseño y verificación del código permite obtener resultados confiables incluso en entornos de bajo nivel como el ensamblador.

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.