Laboratorio 4:
Código ensamblador: ARMv4

José Eduardo Campos Salazar
Escuela Ingeniería en Computadores
Tecnológico de Costa Rica
Cartago, Costa Rica
josc4188@gmail.com

***Resumen*—This report presents the development and implementation of three problems using ARM Assembly language, executed and tested in a digital design environment. The first problem modifies an array in memory based on a conditional operation with a constant. The second problem calculates the factorial of a given number using a loop-based algorithm. The third problem demonstrates input handling by modifying a counter based on specific key inputs. Each case was validated with simulation results, illustrating the correct execution of memory operations, arithmetic instructions, and control flow. The report highlights the importance of low-level programming in understanding hardware-software interaction.**

***Index Terms*—ARM Assembly, factorial calculation, conditional operations, memory access, input handling, digital design, low-level programming.**

I. Introducción

En esta Sección se exploran los fundamentos de la arquitectura ARMv4, abordando sus tipos de instrucciones, el conjunto de registros, el funcionamiento del branch, la implementación de estructuras condicionales, la conversión de código ensamblador a binario y el concepto de endianness. Cada uno de estos aspectos es esencial para comprender el funcionamiento interno de los procesadores ARM, ampliamente utilizados en sistemas embebidos.

1. Tipos de instrucciones en ARMv4

Los tipos de instrucciones que existen en ARMv4 son: [1]

1. **Data processing instructions:** permiten realizar operaciones aritméticas y lógicas sobre valores en los registros, modificando directamente los datos.
   Ejemplos: `ADD`, `SUB`, `CMP`
   Uso:
   ```
   ADD r0, r1, r2   ; r0 = r1 + r2
   ```

2. **Data movement instructions:** mueven datos entre los registros y la memoria en ARM. Estas incluyen:
   - *Single register load and store instructions*
   - *Multiple register load and store instructions*
   - *Single register swap instruction*

   Ejemplos: `LDR`, `STR`, `SWP`
   Uso:
   ```
   LDR r0, [r1]      ; r0 := mem32[r1]
   ```

3. **Control flow instructions:** determinan la siguiente instrucción a ejecutar, habilitando condicionales, bucles y subrutinas.
   Ejemplos: `BEQ`, `BNE`, `BL`
   Uso:
   ```
   CMP r0, #5         ; if (r0 != 5){
   BE BYPASS
   ADD r1, r1, r0     ; r1 = r1 + r0
   SUB r1, r1, r2     ; r1 = r1 - r2}
   BYPASS...
   ```

4. **Special instructions:** controlan aspectos específicos del procesador, como acceso a registros de estado, cambio de modo o manejo de interrupciones.
   Ejemplos: `MRS`, `MSR`
   Uso:
   ```
   MRS r0, CPSR  ; Copia el estado
   del procesador a r0
   MSR CPSR, r1  ; Modifica el estado con
   el contenido de r1
   ```

2. Set de registros en ARM

El set de registros de ARM contiene 16 registros de acceso directo (r0 a r15). Además, el `CPSR` contiene los *condition flags* y los bits de modo actuales.

- Los registros `r0` a `r13` son de propósito general.
- `r14` (Link Register, LR): se usa como "subroutine Link Registerz almacena una copia de `r15` cuando se ejecuta una instrucción `BL`.
- `r15` (Program Counter, PC): mantiene la dirección de la siguiente instrucción a ejecutar. Los bits [1:0] son indefinidos e ignorados; los bits [31:2] contienen el valor del PC.
- Por convención, `r13` se utiliza como el `Stack Pointer` (SP). [2]

3. Funcionamiento del branch

Cuando un programa necesita desviarse de su secuencia de instrucciones por defecto, se utilizan instrucciones de

control de flujo. Estas modifican explícitamente el `Program Counter (PC)`. La forma más simple de control de flujo son las instrucciones **branch** o **jump**. [1]

4. Implementación de condicionales (if/else)

Los condicionales en ARMv4 se implementan mediante el uso del `condition code register`, un registro especial que guarda el estado del resultado de una operación (por ejemplo: cero, negativo, overflow, acarreo, etc.). Posteriormente, se utilizan instrucciones de branch condicionadas que actúan según los bits del registro de condición. [1]

5. Transformación de código ensamblador a binario

Para traducir instrucciones a binario en ARM se emplea el **A32 instruction set encoding**, el cual consta de los siguientes campos: [3]

- **Cond** (4 bits): determina la condición de ejecución. Ejemplo: `0000` = `EQ`, `0001` = `NE`.
- **Bits [27:26]**: suelen ser `00` para instrucciones de procesamiento de datos.
- **I** (bit 25): indica si `Operand2` es un valor inmediato o un registro.
- **Opcode** (bits [24:21]): operación a realizar. Ejemplos: `0100` = `ADD`, `0010` = `SUB`.
- **S** (bit 20): indica si se actualizan las banderas del programa.
- **Rn** (bits [19:16]): primer operando (registro fuente).
- **Rd** (bits [15:12]): destino donde se almacena el resultado.
- **Operand2** (bits [11:0]): segundo operando.

Una instrucción como:

```
ADD r0, r1, r2
```

Se traduce a binario como:

```
1110 00 0 0100 0 0001 0000 0000 0000 0010
```

Herramientas utilizadas para este proceso:

- **GNU Assembler (GAS)**
- **ARM Compiler**

6. Endianness: Little endian vs Big endian

- **Big-endian (BE):** guarda el byte más significativo primero, es decir, el byte más a la izquierda.
- **Little-endian (LE):** guarda el byte menos significativo primero.

Cuando las instrucciones se almacenan en memoria, el orden de los bytes puede causar problemas si se interpreta incorrectamente, especialmente cuando diferentes sistemas con distinta endianness intentan comunicarse o compartir datos binarios. [4]

II. Desarrollo

II-A. Problema 1

Este programa recorre un arreglo de 10 enteros y modifica cada elemento dependiendo de una constante. Si el valor del elemento es mayor o igual a la constante, se multiplica por ella; si es menor, se le suma dicha constante. Este comportamiento se basa en el siguiente algoritmo:

Pasos del algoritmo:

1. Inicializar el índice en 0.
2. Verificar si el índice es igual a 10.
3. Si lo es, finalizar el programa.
4. Si no lo es, leer el valor del arreglo en la posición del índice.
5. Comparar el valor con una constante predefinida.
6. Si el valor es mayor o igual a la constante, multiplicarlo por la constante y almacenarlo de nuevo en el arreglo.
7. Si es menor, sumarle la constante y almacenar el nuevo valor en el arreglo.
8. Incrementar el índice y repetir el proceso desde el paso 2.

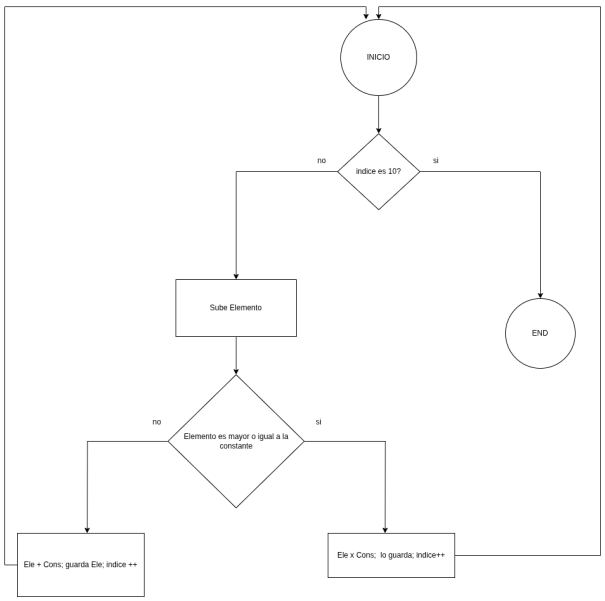

Figura 1. Algoritmo de recorrido y modificación de arreglo

Bloques funcionales utilizados:

- **Contador (índice):** Registra la posición actual dentro del arreglo.
- **Memoria de datos:** Almacena el arreglo de 10 enteros.
- **Comparador:** Compara el valor del elemento con la constante.
- **Unidad aritmética:** Realiza operaciones de suma o multiplicación según el caso.
- **Control de flujo:** Maneja las bifurcaciones del algoritmo y el bucle principal.

II-B. Problema 2

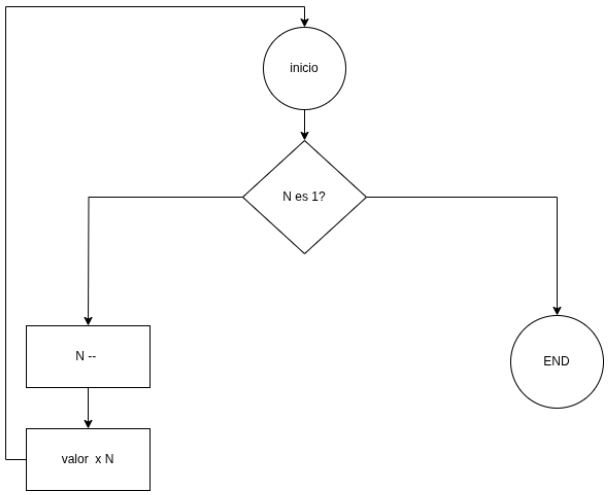

Figura 2. Operación de resta en la FPGA

Se realizó un algoritmo para procesar el factorial de un número X. El algoritmo toma un valor inicial X y lo multiplica sucesivamente por $(X-1)$ hasta llegar a 1. El resultado se almacena en el registro `r2`.

Pasos del algoritmo:

1. Inicializar el registro `r0` con el valor de entrada X.
2. Copiar el valor de `r0` al registro `r2`, que almacenará el resultado del factorial.
3. Comparar si el valor de `r0` es menor o igual a 1.
4. Si lo es, terminar el algoritmo.
5. Si no, decrementar el valor de `r0` en 1.
6. Multiplicar el valor actual de `r0` por `r2`, y almacenar el resultado nuevamente en `r2`.
7. Repetir el proceso desde el paso 3.

Bloques funcionales utilizados:

- **Registro de entrada:** Para cargar el valor inicial X (registro `r0`).
- **Comparador:** Compara si el valor actual es igual o menor a 1 (`cmp r0, #1`).
- **Unidad aritmética:**
  - Para la resta (`sub r0, r0, #1`).
  - Para la multiplicación (`mul r2, r2, r0`).
- **Registro acumulador:** Almacena el resultado parcial en `r2`.
- **Control de flujo:** Ciclo mediante instrucción `b LOOP` y condición de parada con `ble END`.

II-C. Problema 3

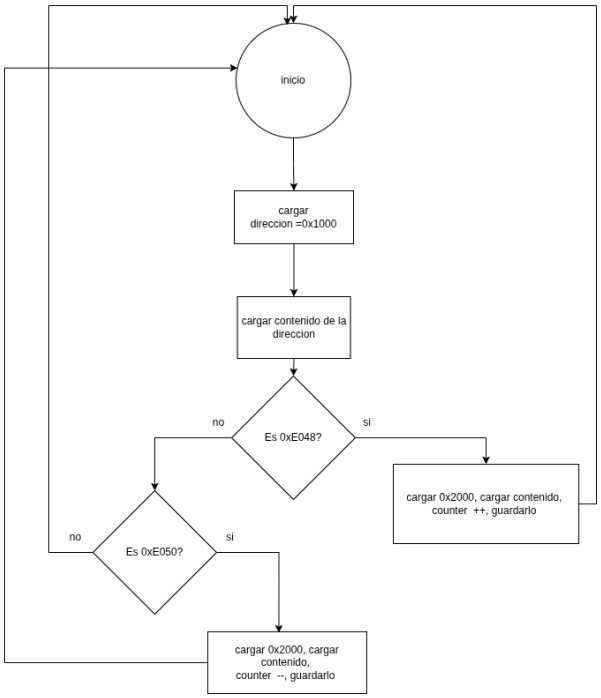

Figura 3. Control de contador mediante botones en la FPGA

Este programa permite incrementar o decrementar un contador dependiendo de si se presiona el botón arriba o abajo en la FPGA o teclado. Se usa lectura de memoria mapeada para detectar la tecla presionada y modificar el valor del contador en consecuencia.

Pasos del algoritmo:

1. Cargar la dirección de memoria donde está almacenado el contador (`0x2000`).
2. Reiniciar el valor del contador a 0.
3. Cargar los códigos asociados a los botones arriba y abajo.
4. Entrar en un ciclo infinito donde:
   a) Se lee la dirección de la tecla presionada (`0x1000`).
   b) Se compara el valor leído con el código de la tecla arriba.
   c) Si coincide, se incrementa el contador.
   d) Si no, se compara con el código de la tecla abajo.
   e) Si coincide, se decrementa el contador.
   f) Si no coincide con ninguno, se vuelve a leer.

Bloques funcionales utilizados:

- **Bloque de entrada:**
  - Dirección de la tecla presionada (`0x1000`).
  - Códigos de teclas arriba (`0xE048`) y abajo (`0xE050`).
- **Comparador:** Compara el valor leído con los códigos de las teclas.
- **Contador:**

- Dirección de contador en `0x2000`.
- Incrementa o decrementa según la tecla presionada.

- **Unidad aritmética:** Realiza las operaciones de suma y resta sobre el contador.
- **Control de flujo:** Usa saltos condicionales para actuar según la tecla presionada y volver al ciclo.

III. Resultados

III-A. Problema 1

En este problema, se procesó un arreglo de 10 enteros donde cada elemento fue modificado según una constante (5). Si el valor era mayor o igual a la constante, se multiplicó por ella; si era menor, se le sumó.

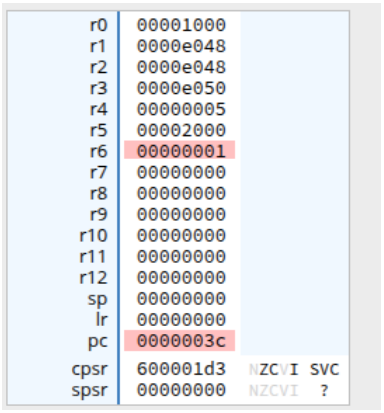

Figura 4. Suma de la constante al primer elemento del arreglo (posición 0)

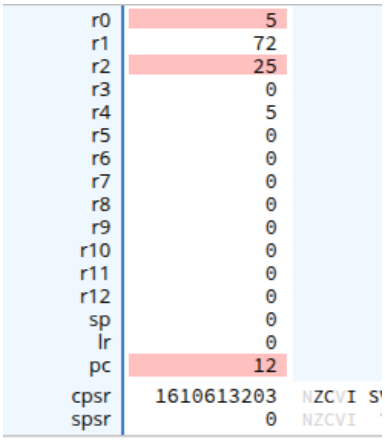

Figura 5. Multiplicación del elemento en la posición 4 por la constante

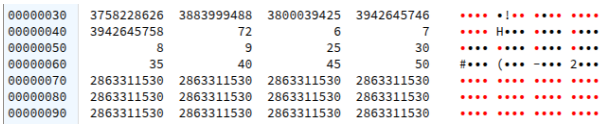

Figura 6. Resultado final del arreglo después del procesamiento

III-B. Problema 2

En el segundo problema, se implementó un algoritmo para calcular el factorial de un número ingresado. El resultado fue almacenado en el registro `r2`.

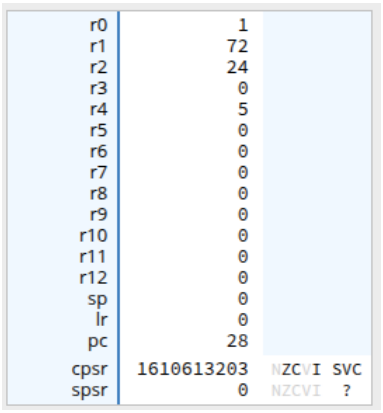

Figura 7. Resultado del factorial de 4 ($4! = 24$) almacenado en `r2`

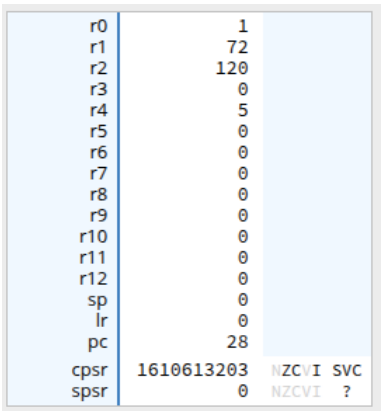

Figura 8. Resultado del factorial de 5 ($5! = 120$) almacenado en `r2`

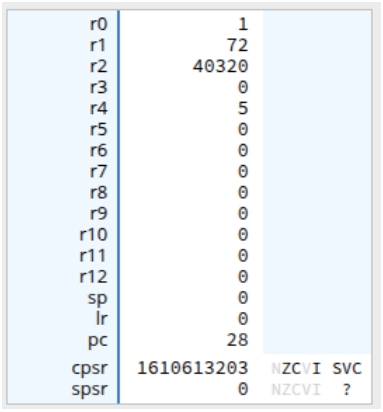

Figura 9. Resultado del factorial de 8 ($8! = 40320$) almacenado en `r2`

III-C. Problema 3

En este caso, se desarrolló un programa que incrementa o decrementa un contador dependiendo de la tecla presionada.

Se utilizaron los códigos de teclas para determinar la acción a realizar.

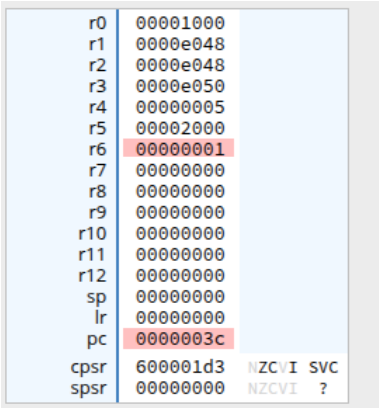

Figura 10. Primera iteración: incremento del contador

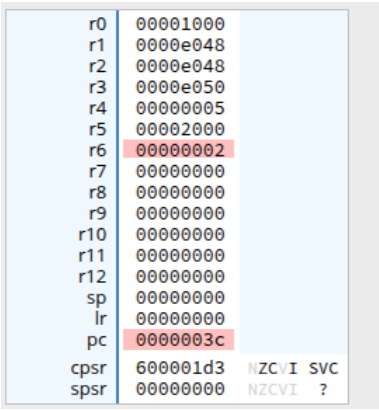

Figura 11. Segunda iteración: nuevo incremento del contador

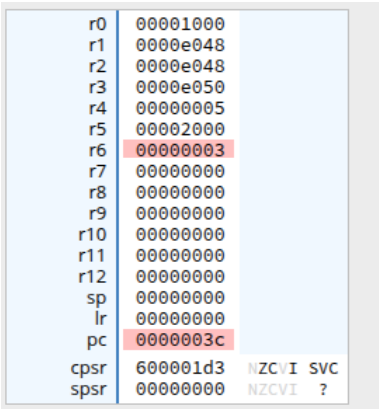

Figura 12. Tercera iteración: el contador continúa incrementando

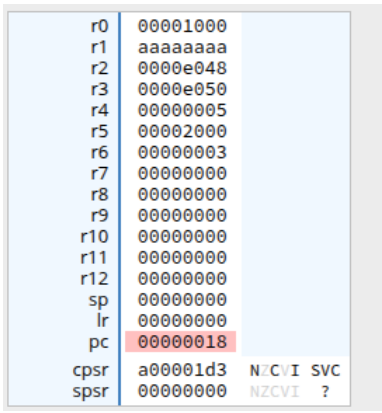

Figura 13. Cuarta iteración: se presiona una tecla no válida, el contador no cambia

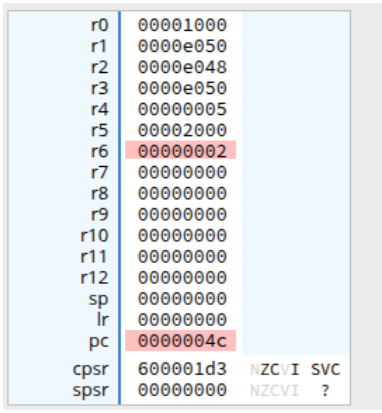

Figura 14. Quinta iteración: decremento del contador

IV. Conclusiones

La resolución de los tres problemas permitió aplicar de forma efectiva los fundamentos del lenguaje ensamblador ARM. Se logró manipular directamente registros, memoria y estructuras de control, consolidando habilidades clave en programación de bajo nivel.

En el primer problema, se implementó exitosamente la modificación condicional de un arreglo, reforzando el uso de comparaciones y acceso indexado. El segundo problema demostró la eficiencia del ensamblador para cálculos matemáticos, como el factorial, utilizando bucles controlados por registros. En el tercer problema, se integró el manejo de entradas externas para modificar un contador, lo que evidenció la interacción directa entre software y hardware mediante direcciones de memoria mapeadas.

En conjunto, la práctica fortaleció la comprensión de la lógica condicional, las operaciones aritméticas básicas y el control de flujo en ensamblador, además de destacar su relevancia en el desarrollo de sistemas embebidos.

Referencias

[1] S. Furber, *ARM System-on-Chip Architecture*, 2nd ed. Addison-Wesley, 2000.

[2] ARM Ltd., “Arm7tdmi technical reference manual - programmer’s model: Registers,” 2000, accessed: 2025-05-10. [Online]. Available: https://developer.arm.com/documentation/ddi0210/c/Programmer-s-Model/Registers?lang=en
[3] Arm Ltd., *A32 Instructions by Encoding*, Arm Developer, mar 2025, accessed: 2025-05-10. [Online]. Available: https://developer.arm.com/documentation/ddi0597/2025-03/A32-Instructions-by-Encoding?lang=en
[4] GeeksforGeeks. (2023) Little and big endian mystery. Accessed: 2025-05-10. [Online]. Available: https://www.geeksforgeeks.org/little-and-big-endian-mystery/