# Tarea 4 INF245:
# Implementación de algoritmos mediante el lenguaje ensamblador ARM Assembly

10 de julio de 2022

**Integrantes:**
**Iván Cano ROL: 202073543-2**
**Claudio Inal ROL: 201873060-2**

# 1. Resumen

En este informe se detallará el procedimiento para la implementación de los algoritmos propuestos en el enunciado de esta tarea, ello mediante el uso del lenguaje ensamblador ARM Assembly y con el simulador QtARMSim. El primer y segundo algoritmo consisten en el cálculo de el tercer lado de un triángulo rectángulo, el primer caso cuando se tienen dos catetos, y el segundo para cuando se tienen un cateto y una hipotenusa. El tercer algoritmo es para el cálculo de la distancia entre dos puntos en el plano cartesiano y el cuarto algoritmo consiste en el cálculo del promedio de un arreglo de números de un largo definido.
Estas soluciones entregarán resultados con números enteros sin decimal, es decir truncados a la unidad.

Los resultados de este trabajo se presentan a continuación:

$$\frac{Resultados\ exitosos}{Resultados\ totales} = \frac{9}{9} = 100\,\%$$

# 2. Introducción

Como se mencionó anteriormente en el resumen, el objetivo de este informe es implementar los algoritmos dados más arriba en lenguaje ARM Assembly, de un modo más cercano a lo que serían los lenguajes de programación de bajo nivel como C, es decir, manteniendo modularidad en las instrucciones y de forma secuencial/iterativa teniendo en cuenta que debemos entregar resultados enteros sin decimal.
Separaremos las instrucciones modularizando de la siguiente manera:

- MAIN: carga los valores guardados en .data y selecciona el modo de acuerdo al enunciado de la tarea.
- MODO1: calcula la hipotenusa de un triángulo rectángulo teniendo los dos catetos.
- MODO2: calcula el tercer cateto de un triángulo rectángulo teniendo los otros dos lados.
- MODO3: calcula la distancia entre dos puntos en el plano cartesiano.
- MODO4: calcula el promedio de un arreglo de números enteros.
- RAIZ: calcula la raíz de un número, entregando un entero truncado a la unidad.

Cuando nos referimos a una forma secuencial es que simularemos el cambio del valor de las variables en registros de forma explícita con las operaciones ya implementadas en ARM Assembly, e iterativamente de modo que podamos simular ciclos while/for con los branch que dispone el lenguaje ensamblador. Un ejemplo de esto se puede ver en el cálculo del promedio de un arreglo por ejemplo, ya que para realizar esta tarea debemos iterar secuencialmente sobre el arreglo sumando uno a uno los elementos, para luego dividir esta suma por el largo del arreglo que los contenía originalmente.
Otro ejemplo es el uso de una técnica para encontrar la raíz de un número cuyo resultado está truncado a la unidad, en donde debemos encontrar un numero cuyo cuadrado perfecto sea el primero inmediatamente menor al número al cual estamos buscando la raíz, esto lo conseguimos iterando sobre un número entero mediante un ciclo while, incrementando el valor de este hasta que se cumplan las condiciones.

A continuación se muestra el detalle del desarrollo de las instrucciones propuestas.

# 3. Desarrollo

Para una mejor organización del informe separaremos las instrucciones (componentes) del programa en subsecciones, estas instrucciones pueden (o no) invocar otras para poder completar su función.

## 3.1. Instrucciones

### 3.1.1. MAIN:

Carga en los registros R4, R5 y R1 el arreglo, largo y el modo desde .data respectivamente. Luego realiza las comparaciones necesarias entre el modo (en R1) para ejecutar el que corresponda:

```
MAIN:
        ldr R4, =arreglo
        ldr R5, =largo
        ldr R1,=modo
        mov R7,#04
        ldr R6, [R5]
        ldr R0,[R1]

        cmp R0,#02
        blt MODO1
        beq MODO2
        cmp R0,#03
        beq MODO3
        bgt MODO4
        bgt MODO4
```

Figura 1: Main

### 3.1.2. MODO1:

Sean $a$ y $b$ los catetos de un triángulo rectángulo, MODO1 calcula la hipotenusa de este triángulo rectángulo con $h = \sqrt{a^2 + b^2}$. Primero carga en R0 la dirección del primer elemento (el cateto) del arreglo, y carga en R1 el segundo elemento con un offset de 4 desde la base del arreglo. Obtiene los cuadrados de estos elementos guardándolos donde estaban originalmente con la operación MUL, luego añade estos dos elementos guardándolos en R0 con la operación ADD, luego hace un branch incondicional a RAIZ:

```
MODO1:
        ldr R0, [R4]
        ldr R1, [R4,#4]
        mul R0,R0,R0
        mul R1,R1,R1
        add R0,R0,R1
        b RAIZ
```

Figura 2: MODO1

### 3.1.3. MODO2:

Sean $a$ y $b$ un cateto y una hipotenusa de un triángulo rectángulo, MODO2 calcula $c = \sqrt{a^2 - b^2}$ si $a$ es la hipotenusa y $c = \sqrt{b^2 - a^2}$ si $b$ es la hipotenusa. Primero carga en R0 la dirección del primer elemento (cateto o hipotenusa), y carga en R1 el segundo elemento con un offset de 4 desde la base del arreglo. Obtiene los cuadrados de estos elementos guardándolos donde estaban originalmente con la operación MUL, luego hace una comparación entre R0 y R1, del cual, dependiendo de las flags hará un branch condicional hacia R0MR1 (R0: cateto < R1: hipotenusa) o R1MR0 (R1: cateto < R0: hipotenusa), dentro de estos branch se realiza la respectiva sustracción entre estos valores de modo que el resultado sea positivo. Luego se hace un branch incondicional hacia RAIZ:

```
MODO2:
        ldr R0, [R4]
        ldr R1, [R4,#4]
        mul R0,R0,R0
        mul R1,R1,R1
        cmp R0,R1
        bgt R0MR1
        blt R1MR0

R0MR1:
        sub R0,R0,R1
        b RAIZ
R1MR0:
        sub R0,R1,R0
        b RAIZ
```

Figura 3: MODO2 e instrucciones auxiliares

### 3.1.4. MODO3:

Sean $(a_x, a_y)$ y $(b_x, b_y)$ puntos en el plano, MODO3 calcula la distancia entre estos puntos en el plano con la fórmula $d = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$. Primero carga los valores $a_x, a_y, b_x, b_y$ en R0, R1, R2 y R3 respectivamente. Sustrae $b_x$ de $a_x$ guardando en R0, luego calcula el cuadrado de este valor con MUL guardándolo en si mismo y reemplazándolo en R2. Sustrae $b_y$ de $a_y$ guardando en R0, luego calcula el cuadrado de este valor con MUL guardándolo en si mismo y reemplazándolo en R3. Añade R2 y R3 en R0 e invoca por medio de branch incondicional a RAIZ para calcular la raíz de R0:

```
MODO3:
        mov R5,#00
        ldr R0, [R4]
        add R5,R5,#04
        ldr R1, [R4,R5]
        add R5,R5,#04
        ldr R2, [R4,R5]
        add R5,R5,#04
        ldr R3, [R4,R5]
        sub R0,R2,R0
        mul R0,R0,R0
        mov R2,R0
        sub R0,R3,R1
        mul R0,R0,R0
        mov R3,R0
        add R0,R2,R3
        b RAIZ
```

Figura 4: MODO3

### 3.1.5. MODO4:

Sea $A_i = \{a_1, a_2, ..., a_n\}$ un arreglo de largo $n$, MODO4 calcula el promedio de este arreglo mediante $\overline{x} = \frac{1}{n} \times \sum_{i=1}^{n} a_i$. Empieza estableciendo $R0 = 0$, luego establece $R7 = 4 \times n$, que corresponde a la dirección objetivo en donde terminarán las siguientes iteraciones: cargar en R1 el elemento con dirección $R4 + R0$ (offset nulo al principio), sumar en R9 el valor en R1 (elemento del arreglo), sumar un offset de 4 al valor de la dirección en el registro R0, comparar con la dirección objetivo; si ya se llegó al objetivo, detenerse, repetir en otro caso.
En este punto realizamos una pequeña comprobación de la suma mostrando el resultado en el LCD del simulador. Guardamos el resultado de R9 (la suma) en R1. Si la suma resulta negativa la multiplicamos por $-1$ en R1. En LOOPDIV simplemente usamos un ciclo que chequea que la diferencia entre el dividendo y el divisor sea positiva, contando en R2 la cantidad de veces que lo sea y finalizando la iteración cuando la diferencia sea negativa. El resultado de la división truncado a la unidad sea encuentra en R2:

```
MODO4:
        mov R0,#00
        mul R7,R7,R6
loopsum:

        ldr R1,[R4,R0]
        add R9,R9,R1
        add R0,R0,#4
        cmp R0,R7
        blt loopsum
.endloopsum:
mov R0,#0
mov R1,#2
mov R2,R9
bl printInt
mov R1, R9
mov R2,#00

cmp R1,#00

bge .ENDDIVNEG
DIVNEG: neg R1,R1
.ENDDIVNEG:
cmp R1,R6
blt .ENDLOOPDIV
#if0
#if+
LOOPDIV:
        sub R1,R1,R6
        add R2,R2,#1
        cmp R1,R6
        bge LOOPDIV

.ENDLOOPDIV:
.ENDMODO4:
```

Figura 5: MODO4

### 3.1.6. RAIZ:

El algoritmo de RAIZ consiste en encontrar el primer cuadrado perfecto inmediatamente menor al número a encontrarle la raiz, de esta forma nos aseguramos de encontrar un número entero sin decimales. Cabe recalcar que esta instrucción se ejecuta entre MODO3 y MODO4, ya que MODO4 es la única instrucción que no hace uso de RAIZ, si es ejecutada, hace un branch incondicional hacia el final de MODO4, de esta manera solo se ejecuta para las instrucciones MODO1, MODO2 y MODO3:

```
RAIZ:
        mov R2, #128
        mov R3,#00
LRAIZ:
        asr R2,R2,#1
        add R3,R3,R2
        mov R1,R3
        mul R1,R1,R1
        cmp R2,#00
        beq .ENDRAIZ
        cmp R1,R0
        beq .ENDRAIZ
        bgt UNSET
        blt LRAIZ
UNSET:
        sub R3,R3,R2
        b LRAIZ
.ENDRAIZ:
        mov R2,R3
        b .ENDMODO4
```

Figura 6: RAIZ

## 3.2. Output y .data

En el caso de .data va tal como se muestra en la Figura 7 (por ejemplo, en modo 3), con modo: un int entre 1 y 4, largo: un int positivo, y arreglo: una lista de int separadas por coma, esto va al principio del programa. Para obtener el resultado en la pantalla LCD de QtARMSim es necesario establecer los registros R0 y R1, además del branch and link printInt y la finalización del programa en la instrucción end como se muestra en la Figura 8, esto va al final del programa:

```
.data
modo: .int 3
largo: .int 4
arreglo: .4byte 0,0,5,5
.text
```

Figura 7: .data

```
mov R0,#0
mov R1,#5
bl printInt

end:    wfi
        .end
```

Figura 8: output

# 4. Resultados

Debido a que los casos de prueba mostrados en el enunciado de la tarea demuestran pasar la condición de ser casos borde (casos elegidos a dedo para intentar hacer fallar al programa) es que haremos uso de los mismos, con los resultados puestos en la siguiente tabla:

| Modo | Largo | Arreglo | Output | Output esperado |
|---|---|---|---|---|
| 1 | 2 | 3, 4 | 5 | 5 |
| 1 | 2 | 3, 5 | 5 | 5 |
| 2 | 2 | 3, 5 | 4 | 4 |
| 2 | 2 | 3, 10 | 9 | 9 |
| 3 | 4 | 0, 0, 5, 5 | 7 | 7 |
| 3 | 4 | 6, 9, 6, 9 | 0 | 0 |
| 4 | 4 | 1, 2, 3, 4 | 5 | 5 |
| 4 | 1 | 14 | 14 | 14 |
| 4 | 5 | -10, -6, 4, 5, 7 | 0 | 0 |

# 5. Análisis

Podemos notar que todos los outputs en LCD de este programa coinciden con los outputs esperados desde el enunciado de la tarea en el apartado de Datos de ejemplo. Consideramos que no fue necesaria una prueba excesivamente extensa para revocar la validez de este programa, teniendo en cuenta que para el arreglo de datos solo tenemos 4 bits en cada elemento, y en ocasiones podríamos generar overflow, pero incluso dejando ese potencial fallo de lado el programa resulta eficiente para cantidades razonables. A continuación se muestran el ratio de resultados exitosos sobre resultados totales:

$$\frac{Resultados\ exitosos}{Resultados\ totales} = \frac{9}{9} = 100\,\%$$

# 6. Conclusiones

Concluimos que se cumple el objetivo de implementar los algoritmos propuestos en el enunciado de la tarea en forma de instrucciones en el lenguaje ARM Assembly. Aunque con algunas dificultades como la de no tener un método para chequear errores de forma eficiente, además del debugger que viene incorporado en QtARMSim para poder testear posibles ciclos infinitos u overflows. Notamos también la similitud entre el lenguaje ensamblador con lenguajes de programación de bajo nivel como lo es C, y el contraste que existe con lenguajes de programación de alto nivel, los cuales tienen un muchísimo nivel de abstracción para la realización de operaciones, a diferencia del lenguaje ensamblador en donde todo se debe implementar.

El nivel de finalización de esta tarea a nuestro criterio es 100 %.