CAPÍTULO 3

Instrucciones de control de flujo

Índice

La mayor parte de los programas mostrados en los capítulos anteriores constaban de una serie de instrucciones que se debían ejecutar una tras otra, siguiendo el orden en el que se habían escrito, hasta que el programa finalizaba. Este tipo de ejecución, en el que las instrucciones se ejecutan una tras otra, siguiendo el orden en el que están en memoria, recibe el nombre de *ejecución secuencial*.

La ejecución secuencial presenta bastantes limitaciones. Un programa de ejecución secuencial no puede, por ejemplo, tomar diferentes acciones en función de los datos de entrada, o de los resultados obtenidos, o de la interacción con el usuario. Tampoco puede repetir un número de veces ciertas operaciones, a no ser que el programador haya repetido varias veces las mismas operaciones en el programa. Pero incluso en ese

caso, el programa sería incapaz de variar el número de veces que dichas instrucciones deberían ejecutarse.

Debido a la gran ventaja que supone el que un programa pueda tomar diferentes acciones y que pueda repetir un conjunto de instrucciones un determinado número de veces, los lenguajes de programación proporcionan estructuras de control que permiten llevar a cabo dichas acciones: las estructuras de control condicionales y repetitivas. Estas estructuras de control permiten modificar el flujo secuencial de instrucciones. En particular, las estructuras de control condicionales permiten la ejecución de ciertas partes del código en función de una serie de condiciones, mientras que las estructuras de control repetitivas permiten la repetición de cierta parte del código hasta que se cumpla una determinada condición de parada.

Este capítulo está organizado como sigue. El Apartado 3.1 describe el registro CCR de ARM, ya que todas las instrucciones de control de flujo condicional de ARM utilizan el contenido de dicho registro para determinar qué acción tomar. El Apartado 3.2 muestra qué son y para qué se utilizan los saltos incondicionales y condicionales. El Apartado 3.3 describe las estructuras de control condicionales *if-then* e *if-then-else*. El Apartado 3.4 presenta las estructuras de control repetitivas *while* y *for*. El último apartado propone una serie de ejercicios más avanzados relacionados con el contenido de este capítulo.

Para complementar la información mostrada en este capítulo y obtener otro punto de vista sobre este tema, se puede consultar el Apartado 3.6 «*ARM's Flow Control Instructions*» del libro «*Computer Organization and Architecture: Themes and Variations*» de Alan Clements. Conviene tener en cuenta que en dicho apartado se utiliza el juego de instrucciones ARM de 32 bits y la sintaxis del compilador de ARM, mientras que en este libro se describe el juego de instrucciones Thumb de ARM y la sintaxis del compilador GCC.

3.1. El registro CCR

Antes de ver cómo se implementan las estructuras de control condicionales y repetitivas, es necesario explicar que cada vez que se ejecuta una instrucción aritmética, se actualiza el *Condition Code Register*[1] (CCR). Por ejemplo, al ejecutar una instrucción de suma «**add**», se actualiza el indicador de acarreo C (que se corresponde con un bit concreto del

[1] Los distintos fabricantes de computadores llaman de forma distinta al registro en el que se almacena el estado del computador después de la ejecución de una operación. Por ejemplo, ARM llama a este registro *Current Processor Status Register* (CPSR), mientras que Intel lo denomina *Status Register* (SR). Nosotros seguiremos la notación utilizada en [Cle14], que llama a este registro *Condition Code Register* (CCR).

registro CCR). Así mismo, al ejecutar la instrucción «**cmp**», que compara dos registros, se actualiza, entre otros, el indicador de cero Z (que se corresponde con otro de los bits del registro CCR). Los valores de los indicadores (*flags*, en inglés) del registro CCR son usados por las instrucciones de salto para decidir qué camino debe tomar el programa.

Los cuatro indicadores del registro CCR que se utilizan por las instrucciones de control de flujo son:

N: Se activa[2] cuando el resultado de la operación es negativo (el 0 se considera positivo).

Z: Se activa cuando el resultado de la operación es 0.

C: Se activa cuando el resultado de la operación produce un acarreo (llamado *carry* en inglés).

V: Se activa cuando el resultado de la operación produce un desbordamiento en operaciones con signo (*overflow*).

La instrucción «**cmp** r1, r2» actualiza el registro CCR en función de los valores almacenados en los registros r1 y r2. Para ello, y como se ha visto en el capítulo anterior, realiza la resta r1 − r2 y, dependiendo del resultado obtenido, activa o no los distintos indicadores. Por ejemplo, si suponemos que los registros r1, r2 y r3 contienen los valores 5, 3 y 5, respectivamente, entonces:

- Tras la operación «**cmp** r1, r2», el indicador N no se activará puesto que el resultado de la comparación no es negativo (r1 − r2 es un número positivo). El indicador Z tampoco se activará puesto que los registros contienen valores diferentes (r1 − r2 es distinto de cero).
- Tras la operación «**cmp** r2, r1», el indicador N se activará puesto que el resultado de la comparación produce un número negativo (r2 − r1 es un número negativo). El indicador Z no se activará puesto que los registros contienen valores diferentes.
- Tras la operación «**cmp** r1, r3», el indicador N no se activará puesto que el resultado de la comparación no es negativo (el cero se considera positivo). El indicador Z si se activará puesto que los registros contienen valores iguales (r1 − r3 es igual a cero).

[2] Como ya se sabe, un bit puede tomar uno de dos valores: 0 o 1. Cuando decimos que un bit está activado (*set*), nos referimos a que dicho bit tiene el valor 1. Cuando decimos que un bit está desactivado (*clear*), nos referimos a que dicho bit tiene el valor 0. Además, cuando decimos que un bit se activa (se pone a 1) bajo determinadas circunstancias, se sobreentiende que si no se dan dichas circunstancias, dicho bit se desactiva (se pone a 0). Lo mismo ocurre, pero al revés, cuando decimos que un bit se desactiva bajo determinadas circunstancias.

Los indicadores del registro CCR se muestran en Qt ARMSim en la esquina inferior derecha de la ventana del simulador. La Figura 3.1 muestra un ejemplo donde el indicador N está activo y los otros tres no.

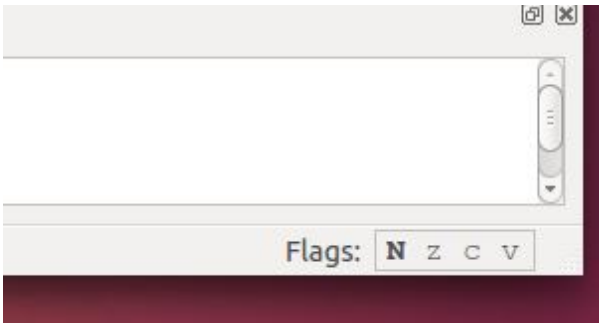

Figura 3.1: Indicadores mostrados por el simulador Qt ARMSim

El siguiente programa muestra un ejemplo en el que se modifican los indicadores del registro CCR comparando el contenido de distintos registros.

cap3-E0.s

```
1         .text
2 main:   mov r1, #10
3         mov r2, #5
4         mov r3, #10
5         cmp r1, r2
6         cmp r1, r3
7         cmp r2, r3
8 stop:   wfi
```

EJERCICIOS

Copia y ensambla el programa anterior. A continuación, ejecuta el programa paso a paso conforme vayas respondiendo las siguientes preguntas:

▶ **3.1** Carga y ejecuta el programa anterior. ¿Se activa el indicador N tras la ejecución de la instrucción «`cmp r1, r2`»? ¿y el indicador Z?

▶ **3.2** ¿Se activa el indicador N tras la ejecución de la instrucción «`cmp r1, r3`»? ¿y el indicador Z?

▶ **3.3** ¿Se activa el indicador N tras la ejecución de la instrucción «`cmp r2, r3`»? ¿y el indicador Z?

3.2. Saltos incondicionales y condicionales

En este apartado se describen las instrucciones de salto disponibles en el ensamblador Thumb de ARM. En primer lugar se verá qué son los saltos incondicionales y a continuación, los saltos condicionales.

3.2.1. Saltos incondicionales

Se denominan saltos incondicionales a aquéllos que se realizan siempre, que no dependen de que se cumpla una determinada condición. La instrucción de salto incondicional es «`b etiqueta`», donde «`etiqueta`» referencia la línea del programa a la que se quiere saltar. Al tratarse de una instrucción de salto incondicional, cada vez que se ejecuta la instrucción «`b etiqueta`», el programa saltará a la instrucción etiquetada con «`etiqueta`», independientemente de qué valor tenga el registro `CCR`.

El siguiente programa muestra un ejemplo de salto incondicional.

cap3-E1.s

```
1         .text
2 main:   mov r0, #5
3         mov r1, #10
4         mov r2, #100
5         mov r3, #0
6         b salto
7         add r3, r1, r0
8 salto:  add r3, r3, r2
9 stop:   wfi
```

EJERCICIOS

▶ **3.4** Carga y ejecuta el programa anterior. ¿Qué valor almacena el registro `r3` al finalizar el programa?

▶ **3.5** Comenta completamente la línea «`b salto`» (o bórrala) y vuelve a ejecutar el programa. ¿Qué valor almacena ahora el registro `r3` al finalizar el programa?

▶ **3.6** ¿Qué pasaría si la etiqueta «`salto`» estuviera situada en la línea «`mov r1, #10`», es decir, antes de la instrucción «`b salto`»?

▶ **3.7** Crea un nuevo código basado en el código anterior, pero en el que la línea etiquetada con «`salto`» sea la línea «`mov r1,#10`». Ejecuta el programa paso a paso y comprueba que lo que ocurre coincide con lo que habías deducido en el ejercicio anterior.

3.2.2. Saltos condicionales

Las instrucciones de salto condicional tiene la forma «**bXX** `etiqueta`», donde «`XX`» se sustituye por un nemotécnico que indica el tipo de condición que se debe cumplir para realizar el salto y «`etiqueta`» referencia a la línea del programa a la que se quiere saltar. Las instrucciones de salto condicional comprueban ciertos indicadores del registro `CCR` para decidir si se debe saltar o no. Por ejemplo, la instrucción «**beq** `etiqueta`» (*branch if equal*) comprueba si el indicador `Z` está activo. Si está activo, entonces salta a la instrucción etiquetada con «`etiqueta`». Si no lo está, el programa continúa con la siguiente instrucción. De forma similar, «**bne** `etiqueta`» (*branch if not equal*) saltará a la instrucción etiquetada con «`etiqueta`» si el indicador `Z` no está activo, si esta activo, no saltará.

El Cuadro 3.1 muestra las distintas instrucciones de salto condicional (en el Cuadro 3.2 de [Cle14] se puede consultar también la codificación en binario asociada a cada instrucción de salto).

Cuadro 3.1: Instrucciones de salto condicional

Instrucción	Condición de salto
«**beq**» (*branch if equal*)	Iguales (`Z`)
«**bne**» (*branch if not equal*)	No iguales (`z`)
«**bcs**» (*branch if carry set*)	Mayor o igual sin signo (`C`)
«**bcc**» (*branch if carry clear)*	Menor sin signo (`c`)
«**bmi**» (*branch if minus*)	Negativo (`N`)
«**bpl**» (*branch if plus*)	Positivo o cero (`n`)
«**bvs**» (*branch if overflow set*)	Desbordamiento (`V`)
«**bvc**» (*branch if overflow clear*)	No hay desbordamiento (`v`)
«**bhi**» (*branch if higher*)	Mayor sin signo (`Cz`)
«**bls**» (*branch if lower of same*)	Menor o igual sin signo (`c` o `Z`)
«**bge**» (*branch if greater or equal*)	Mayor o igual (`NV` o `nv`)
«**blt**» (*branch if less than*)	Menor que (`Nv` o `nV`)
«**bgt**» (*branch if greater than*)	Mayor que (`z` y (`NV` o `nv`))
«**ble**» (*branch if less than or equal*)	Menor o igual (`Nv` o `nV` o `Z`)

El siguiente ejemplo muestra un programa en el que se utiliza la instrucción «**beq**» para saltar en función del resultado de la operación anterior.

cap3-E2.s

```
1         .text
2 main:   mov r0, #5
3         mov r1, #10
4         mov r2, #5
5         mov r3, #0
```

```
6         cmp r0, r2
7         beq salto
8         add r3, r0, r1
9   salto: add r3, r3, r1
10  stop:  wfi
```

EJERCICIOS

▶ **3.8** Carga y ejecuta el programa anterior. ¿Qué valor tiene el registro `r3` cuando finaliza el programa?

▶ **3.9** ¿En qué estado está el indicador `Z` tras la ejecución de la instrucción «**`cmp`** `r0, r2`»? Para contestar a esta pregunta deberás recargar la simulación y detener la ejecución justo después de ejecutar dicha instrucción.

▶ **3.10** Cambia la línea «**`cmp`** `r0, r2`» por «**`cmp`** `r0, r1`» y vuelve a ejecutar el programa. ¿Qué valor tiene ahora el registro `r3` cuando finaliza el programa?

▶ **3.11** ¿En qué estado está el indicador `Z` tras la ejecución de la instrucción «**`cmp`** `r0, r1`»?

▶ **3.12** ¿Por qué se produce el salto en el primer caso, «**`cmp`** `r0, r2`», y no en el segundo, «**`cmp`** `r0, r1`»?

Veamos ahora el funcionamiento de la instrucción «**`bne`** `etiqueta`». Para ello, se usará el programa anterior (sin la modificación propuesta en los ejercicios anteriores), pero en el que deberás sustituir la instrucción «**`beq`** `salto`» por «**`bne`** `salto`».

EJERCICIOS

▶ **3.13** Carga y ejecuta el programa. ¿Qué valor tiene el registro `r3` cuando finaliza el programa?

▶ **3.14** ¿En qué estado está el indicador `Z` tras la ejecución de la instrucción «**`cmp`** `r0, r2`»?

▶ **3.15** Cambia la línea «**`cmp`** `r0, r2`» por «**`cmp`** `r0, r1`» y vuelve a ejecutar el programa. ¿Qué valor tiene ahora el registro `r3` cuando finaliza el programa?

▶ **3.16** ¿En qué estado está el indicador `Z` tras la ejecución de la instrucción «**`cmp`** `r0, r1`»?

▶ **3.17** ¿Por qué no se produce el salto en el primer caso, «**`cmp`** `r0, r2`», y sí en el segundo, «**`cmp`** `r0, r1`»?

3.3. Estructuras de control condicionales

En este apartado se describen las estructuras de control condicionales *if-then* e *if-then-else*.

3.3.1. Estructura condicional *if-then*

La estructura condicional *if-then* está presente en todos los lenguajes de programación y se usa para realizar o no un conjunto de acciones dependiendo de una condición. A continuación se muestra un programa escrito en Python3 que utiliza la estructura *if-then*.

```
X = 1
Y = 1
Z = 0

if (X == Y):
        Z = X + Y
```

El programa anterior comprueba si el valor de `X` es igual al valor de `Y` y en caso de que así sea, suma los dos valores, almacenando el resultado en `Z`. Si no son iguales, `Z` permanecerá inalterado.

Una posible implementación del programa anterior en ensamblador Thumb de ARM, sería la siguiente:

cap3-E3.s

```
        .data
X:      .word 1
Y:      .word 1
Z:      .word 0

        .text
main:   ldr r0, =X
        ldr r0, [r0]    @ r0 <- [X]
        ldr r1, =Y
        ldr r1, [r1]    @ r1 <- [Y]

        cmp r0, r1
        bne finsi
        add r2, r0, r1  @-
        ldr r3, =Z      @ [Z] <- [X] + [Y]
        str r2, [r3]    @-

finsi:  wfi
```

EJERCICIOS

▶ **3.18** Carga y ejecuta el programa anterior. ¿Qué valor tiene la posición de memoria `Z` cuando finaliza el programa?

▶ **3.19** Modifica el código para que el valor de `Y` sea distinto del de `X` y vuelve a ejecutar el programa. ¿Qué valor hay ahora en la posición de memoria `Z` cuando finaliza el programa?

Como has podido comprobar, la idea fundamental para implementar la instrucción «**`if x==y:`**» ha consistido en utilizar una instrucción de salto condicional que salte si no se cumple dicha condición, «**`bne`**». De esta forma, si los dos valores comparados son iguales, el programa continuará, ejecutando el bloque de instrucciones que deben ejecutarse solo si «`x==y`» (una suma en este ejemplo). En caso contrario, se producirá el salto y dicho bloque no se ejecutará.

EJERCICIOS

▶ **3.20** Cambia el programa anterior para que realice la suma cuando X sea mayor a Y.

3.3.2. Estructura condicional *if-then-else*

Sea el siguiente programa en Python3:

```
1 X = 1
2 Y = 1
3 Z = 0
4
5 if (X == Y):
6         Z = X + Y
7 else:
8         Z = X + 5
```

En este caso, si se cumple la condición (¿son iguales `X` y `Y`?) se realiza una acción, sumar `X` y `Y`, y si no son iguales, se realiza otra acción diferente, sumar el número 5 a `X`.

Una posible implementación del programa anterior en ensamblador Thumb de ARM, sería la siguiente:

cap3-E4.s

```
1         .data
2 X:      .word 1
3 Y:      .word 1
4 Z:      .word 0
5
6         .text
7 main:   ldr r0, =X
8         ldr r0, [r0]            @ r0 <- [X]
9         ldr r1, =Y
10        ldr r1, [r1]            @ r1 <- [Y]
```

```
11
12          cmp r0, r1
13          bne else
14          add r2, r0, r1          @ r2 <- [X] + [Y]
15          b finsi
16
17  else:   add r2, r0, #5          @ r2 <- [X] + 5
18
19  finsi:  ldr r3, =Z
20          str r2, [r3]            @ [Z] <- r2
21
22  stop:   wfi
```

EJERCICIOS

▶ **3.21** Carga y ejecuta el programa anterior. ¿Qué valor hay en la posición de memoria `Z` cuando finaliza el programa?

▶ **3.22** Cambia el valor de `Y` para que sea distinto de `X` y vuelve a ejecutar el programa. ¿Qué valor hay ahora la posición de memoria `Z` al finalizar el programa?

▶ **3.23** Supón que el programa anterior en Python3, en lugar de la línea «`if X == Y:`», tuviera la línea «`if X > Y:`». Cambia el programa en ensamblador para se tenga en cuenta dicho cambio. ¿Qué modificaciones has realizado en el programa en ensamblador?

▶ **3.24** Supón que el programa anterior en Python3, en lugar de la línea «`if X == Y:`», tuviera la línea «`if X <= Y:`». Cambia el programa en ensamblador para se tenga en cuenta dicho cambio. ¿Qué modificaciones has realizado en el programa en ensamblador?

3.4. Estructuras de control repetitivas

Una vez vistas las estructuras de control condicionales, vamos a ver ahora las estructuras de control repetitivas *while* y *for*.

3.4.1. Estructura de control repetitiva *while*

La estructura de control repetitiva *while* permite ejecutar repetidamente un bloque de código mientras se siga cumpliendo una determinada condición. La estructura *while* funciona igual que una estructura *if-then*, en el sentido de que si se cumple la condición evaluada, se ejecuta el código asociado al cumplimiento de dicha condición. Pero a diferencia de la estructura *if-then*, una vez se ha ejecutado la última instrucción del

código asociado a la condición, el control vuelve a la evaluación de la condición y todo el proceso se vuelve a repetir.

El siguiente programa en Python3 muestra el uso de la estructura de control repetitiva *while*.

```
X = 1
E = 1
LIM  = 100

while (X<LIM):
    X = X + 2 * E
    E = E + 1
    print(X)
```

El programa anterior realizará las operaciones $X = X + 2 \cdot E$ y $E = E + 1$ mientras se cumpla que $X < LIM$. Por lo tanto, la variable X irá tomando los siguientes valores con cada iteración del bucle: $3, 7, 13, 21, 31, 43, 57, 73, 91, 111$.

Una posible implementación del programa anterior en ensamblador Thumb de ARM sería la siguiente:

cap3-E6.s

```
        .data
X:      .word 1
E:      .word 1
LIM:    .word 100

        .text
main:   ldr r0, =X
        ldr r0, [r0]    @ r0 <- X
        ldr r1, =E
        ldr r1, [r1]    @ r1 <- E
        ldr r2, =LIM
        ldr r2, [r2]    @ r2 <- LIM

bucle:  cmp r0, r2
        bpl finbuc
        lsl r3, r1, #1  @ r3 <- 2 * [E]
        add r0, r0, r3  @ r0 <- [X] + 2 * [E]
        add r1, r1, #1  @ r1 <- [E] + 1
        ldr r4, =X
        str r0, [r4]    @ [X] <- r0
        ldr r4, =E
        str r1, [r4]    @ [E] <- r1
        b   bucle

finbuc: wfi
```

EJERCICIOS

▶ **3.25** ¿Qué hacen las instrucciones «`cmp r0, r2`», «`bpl finbuc`» y «`b bucle`»?

▶ **3.26** ¿Por qué se ha usado «`bpl`» y no «`bmi`»?

▶ **3.27** ¿Qué indicador del registro `CCR` se está comprobando cuando se ejecuta la instrucción «`bpl`»? ¿Cómo debe estar dicho indicador, activado o desactivado, para que se ejecute el interior del bucle?

▶ **3.28** ¿Qué instrucción se ha utilizado para calcular $2 \cdot E$? ¿Qué hace dicha instrucción?

▶ **3.29** En el código mostrado se actualiza el contenido de las variables X y E en cada iteración del bucle. ¿Se te ocurre algún tipo de optimización que haga que dicho bucle se ejecute mucho más rápido? (El resultado final del programa debe ser el mismo.)

3.4.2. Estructura de control repetitiva *for*

En muchas ocasiones es necesario repetir un conjunto de acciones un número predeterminado de veces. Esto podría conseguirse utilizando un contador que se fuera incrementando de uno en uno y un bucle *while* que comprobara que dicho contador no ha alcanzado un determinado valor. Sin embargo, como dicha estructura se da con bastante frecuencia, los lenguajes de programación de alto nivel suelen proporcionar una forma de llevarla a cabo directamente.

El siguiente programa muestra un ejemplo de uso de la estructura de control repetitiva *for* en Python3.

```
1 V = [2, 4, 6, 8, 10]
2 n = 5
3 suma = 0
4
5 for i in range(n):  # i = [0..N-1]
6   suma = suma + V[i]
```

El programa anterior suma todos los valores de un vector V y almacena el resultado en la variable *suma*. Puesto que en dicho programa se sabe el número de iteraciones que debe realizar el bucle (que es igual al número de elementos del vector), la estructura ideal para resolver dicho problema es el bucle *for*. Se deja como ejercicio para el lector pensar en cómo resolver el mismo problema utilizando la estructura *while*.

Por otro lado, un programa más realista en Python3 no necesitaría la variable n, ya que sería posible obtener la longitud del vector utilizando la función «`len(V)`». Si se ha utilizado la variable n ha sido para acercar

el problema al caso del ensamblador, en el que sí que se va a tener que recurrir a dicha variable.

Una posible implementación del programa anterior en ensamblador Thumb de ARM sería la siguiente:

cap3-E5.s

```
1         .data
2  V:     .word 2, 4, 6, 8, 10
3  n:     .word 5
4  suma:  .word 0
5
6         .text
7  main:  ldr r0, =V      @ r0 <- dirección de V
8         ldr r1, =n
9         ldr r1, [r1]    @ r1 <- n
10        ldr r2, =suma
11        ldr r2, [r2]    @ r2 <- suma
12        mov r3, #0      @ r3 <- 0
13
14 bucle: cmp r3, r1
15        beq finbuc
16        ldr r4, [r0]
17        add r2, r2, r4  @ r2 <- r2 + V[i]
18        add r0, r0, #4
19        add r3, r3, #1
20        b bucle
21
22 finbuc: ldr r0, =suma
23        str r2, [r0]    @ [suma] <- r2
24
25 stop:  wfi
```

Como se puede comprobar en el código anterior, el índice del bucle está almacenado en el registro `r3` y la longitud del vector en el registro `r1`. En cada iteración se comprueba si el índice es igual a la longitud del vector. En caso positivo, se salta al final del bucle y en caso negativo, se realizan las operaciones indicadas en el bucle.

EJERCICIOS

▶ **3.30** ¿Para qué se utilizan las siguientes instrucciones: «**`cmp r3, r1`**», «**`beq finbuc`**», «**`add r3, r3, #1`**» y «**`b bucle`**»?

▶ **3.31** ¿Qué indicador del registro `CCR` se está comprobando cuando se ejecuta la instrucción «**`beq finbuc`**»?

▶ **3.32** ¿Qué contiene el registro `r0`? ¿Para qué sirve la instrucción «**`ldr r4, [r0]`**»?

▶ **3.33** ¿Para qué sirve la instrucción «**`add r0, r0, #4`**»?

▶ **3.34** ¿Qué valor tiene la posición de memoria «`suma`» cuando finaliza la ejecución del programa? ¿Y si $V = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ y $n = 10$?

3.5. Problemas del capítulo

EJERCICIOS

▶ **3.35** Implementa un programa que dados dos números almacenados en dos posiciones de memoria A y B, almacene el valor absoluto de la resta de ambos en la posición de memoria RES. Es decir, si A es mayor que B deberá realizar la operación $A - B$ y almacenar el resultado en RES, y si B es mayor que A, entonces deberá almacenar $B - A$.

▶ **3.36** Implementa un programa que dado un vector, calcule el número de elementos de un vector que son menores a un número dado. Por ejemplo, si el vector es $[2, 4, 6, 3, 10, 12, 2]$ y el número es 5, el resultado esperado será 4, puesto que hay 4 elementos del vector menores a 5.

▶ **3.37** Implementa un programa que dada las notas de 2 exámenes parciales almacenadas en memoria (con notas entre 0 y 10), calcule la nota final (sumando las dos notas) y almacene en memoria la cadena de caracteres *APROBADO* si la nota final es mayor o igual a 10 y *SUSPENDIDO* si la nota final es inferior a 10.

▶ **3.38** Modifica el programa anterior para que almacene en memoria la cadena de caracteres *APROBADO* si la nota final es mayor o igual a 10 pero menor a 14, *NOTABLE* si la nota final es mayor o igual a 14 pero menor a 18 y *SOBRESALIENTE* si la nota final es igual o superior a 18.

▶ **3.39** Modifica el programa anterior para que si alguna de las notas de los exámenes parciales es inferior a 5, entonces se almacene *NOSUPERA* independientemente del valor del resto de exámenes parciales.

▶ **3.40** Implementa un programa que dado un vector, sume todos los elementos del mismo mayores a un valor dado. Por ejemplo, si el vector es $[2, 4, 6, 3, 10, 1, 4]$ y el valor 5, el resultado esperado será $6 + 10 = 16$

▶ **3.41** Implementa un programa que dado un vector, sume todos los elementos pares. Por ejemplo, si el vector es $[2, 7, 6, 3, 10]$, el resultado esperado será $2 + 6 + 10 = 18$.

▶ **3.42** Implementa un programa que calcule los N primeros números de la sucesión de Fibonacci. La sucesión comienza con los números 1 y 1 y a partir de estos, cada término es la suma de los dos anteriores. Es decir que el tercer elemento de la sucesión es $1 + 1 = 2$, el cuarto $1 + 2 = 3$, el quinto $2 + 3 = 5$ y así sucesivamente. Por ejemplo, los

$N = 10$ primeros son $[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]$. Para ello deberás usar una estructura de repetición adecuada y almacenar los valores obtenidos en memoria.

▶ **3.43** Modifica el programa anterior para que calcule elementos de la sucesión de Fibonacci hasta que el último elemento calculado sea mayor a un valor concreto.

Bibliografía

[Adv95] Advanced RISC Machines Ltd (ARM) (1995). *ARM 7TDMI Data Sheet*.
URL http://www.ndsretro.com/download/ARM7TDMI.pdf

[Atm11] Atmel Corporation (2011). *ATmega 128: 8-bit Atmel Microcontroller with 128 Kbytes in-System Programmable Flash*.
URL http://www.atmel.com/Images/doc2467.pdf

[Atm12] Atmel Corporation (2012). *AT91SAM ARM-based Flash MCU datasheet*.
URL http://www.atmel.com/Images/doc11057.pdf

[Bar14] S. Barrachina Mir, G. León Navarro y J. V. Martí Avilés (2014). *Conceptos elementales de computadores*.
URL http://lorca.act.uji.es/docs/conceptos_elementales_de_computadores.pdf

[Cle14] A. Clements (2014). *Computer Organization and Architecture. Themes and Variations. International edition.* Editorial Cengage Learning. ISBN 978-1-111-98708-4.

[Shi13] S. Shiva (2013). *Computer Organization, Design, and Architecture, Fifth Edition*. Taylor & Francis. ISBN 9781466585546.
URL http://books.google.es/books?id=m5KlAgAAQBAJ