



# Universidad Nacional Autónoma de México



## Facultad de Ingeniería

### Integrantes:

Espinoza Matamoros Percival Ulises - 320025561

Flores Colin Victor Jaziel - 320266083

Lara Hernandez Angel Husiel - 320060829

## Laboratorio de Microcomputadoras

Grupo: 06 - Semestre: 2026-2

### Practica 1:

Introducción de las arquitecturas ARM empleando  
Raspberry Pi

### Profesor:

Ing. Moises Melendez Reyes

### Fecha de Entrega:

1 de Marzo del 2026

## 1. Objetivo:

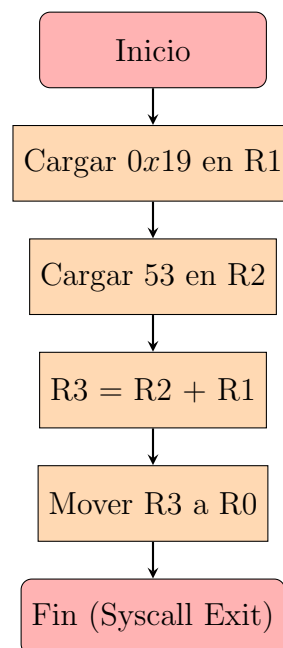
Aprender la estructura de los procesadores con arquitectura ARM, utilizar la plataforma Raspberry Pi, los entornos para programar; desarrollar algoritmos con las instrucciones en lenguaje ensamblador, controlar directamente los recursos del microprocesador; editar, compilar, ensamblar, simular y ejecutar programas en Raspberry Pi.

## Actividad 1

Seguir el procedimiento indicado en el apartado cuarto de manual de tutoriales, escribir, comentar y ensamblar y ejecutar el siguiente programa; explicar qué hace.

## Propuesta de solución

Se propone cargar cada operando directamente en un registro mediante **direccionamiento inmediato** (`MOV Rn, #valor`), de modo que el dato viaja desde la instrucción misma hacia el banco de registros sin pasar por memoria. Una vez que ambos operandos residen en R1 y R2, la ALU calcula la suma con `ADD R3, R2, R1` y deposita el resultado en R3; finalmente, dicho valor se copia a R0 (registro de retorno por convención ABI) antes de invocar la llamada al sistema de salida. El diagrama de flujo que representa este proceso es:



**Desarrollo** Se transcribió y compiló el código fuente. Se utilizó GDB para la depuración y verificación de los registros de la CPU.

Listing 1: Código de la Actividad 1

```
1  .global _start      @ Hace visible la etiqueta _start para el
                        enlazador (linker)
2
3  _start:             @ Punto de entrada del programa
4      MOV R1, #0x19    @ Carga el valor hexadecimal 19 (25 decimal)
                        en el registro R1
5      MOV R2, #53      @ Carga el valor decimal 53 en el registro
                        R2
6
7      ADD R3, R2, R1    @ Suma: R3 = R2 + R1. Resultado esperado: 78
8
9      MOV R0, R3        @ Mueve el resultado (78) al registro R0 (
                        registro de retorno)
10     MOV R7, #1        @ Carga el valor 1 en R7. En Linux, 1
                        significa "Syscall Exit"
11     SVC 0             @ Supervisor Call: Llama al Kernel para
                        ejecutar la salida
```

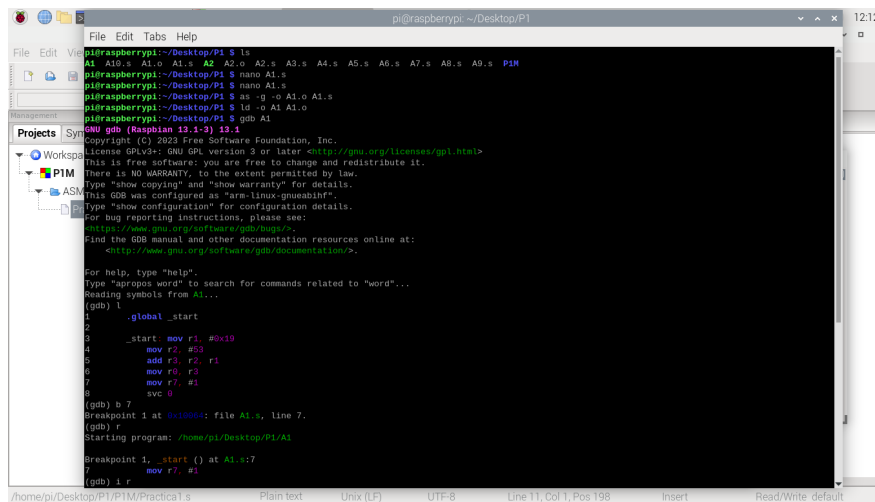


Figura 1: Proceso de ensamblado, enlazado e inicio de GDB en Raspberry Pi.

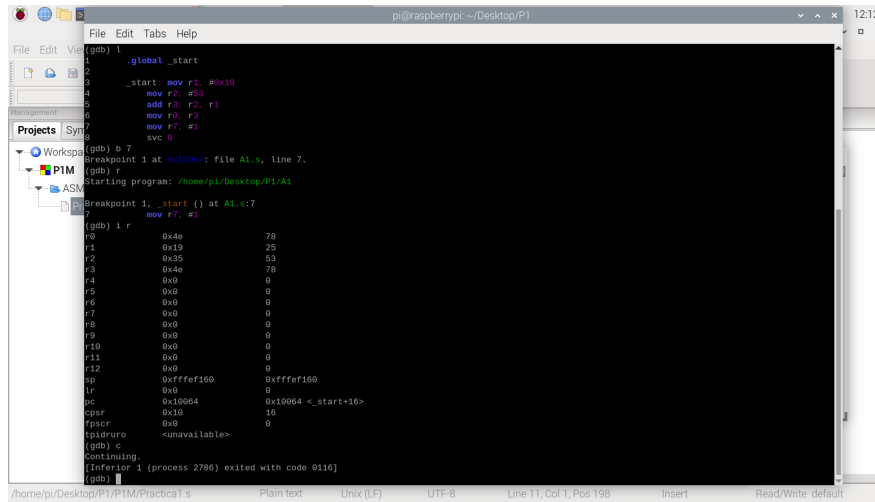


Figura 2: Inspección de registros en GDB mostrando R0 y R3 con el valor de 78 (0x4E).

## Análisis de resultados

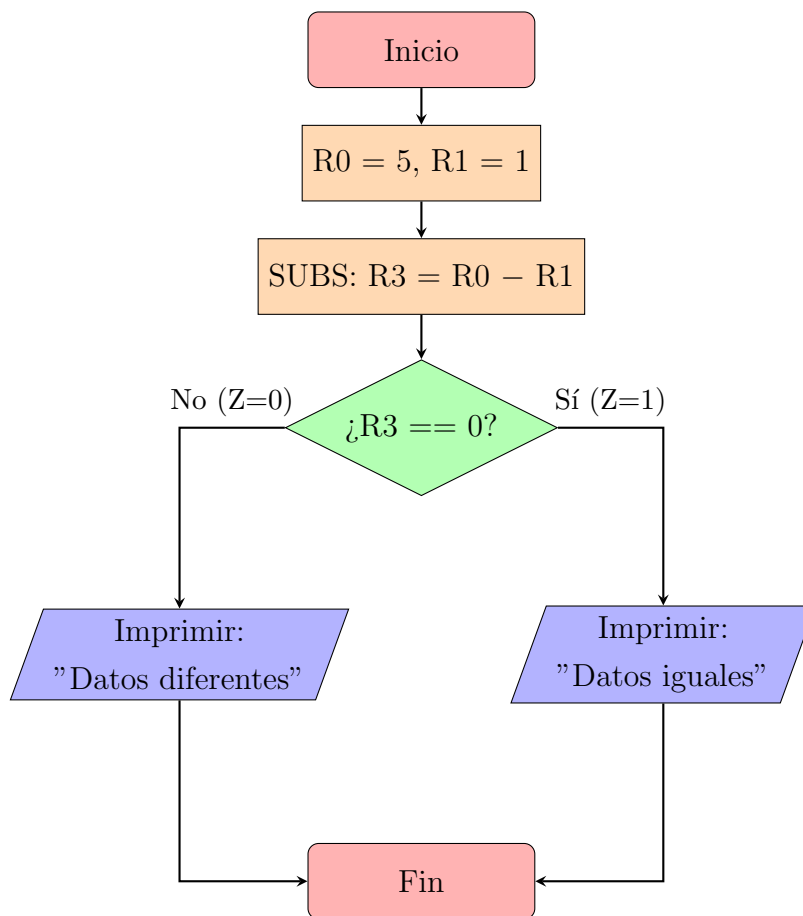
El programa utiliza el **modo de direccionamiento inmediato** para cargar constantes estáticas en los registros internos del procesador (`MOV R1, #0x19` y `MOV R2, #53`). El flujo interno de los datos envía el contenido de los registros R1 y R2 a la Unidad Aritmético Lógica (ALU) a través de la instrucción `ADD`, la cual deposita el resultado de la suma matemática en el registro R3. Se comprobó que el flujo y la lógica son correctos, ya que la suma de  $25 + 53$  arroja 78 en decimal, equivalente a `0x4E` en hexadecimal, valor que se observa claramente almacenado en los registros durante la ejecución controlada con GDB.

## Actividad 2

Seguir el procedimiento indicado en el apartado cuarto de manual de tutoriales, escribir, comentar, ensamblar y ejecutar el siguiente programa; explicar qué hace.

### Propuesta de solución

El problema requiere comparar dos valores y seleccionar un mensaje de salida según el resultado. Se propone usar la instrucción **SUBS** para restar ambos valores y actualizar simultáneamente la bandera **Zero (Z)** del registro **CPSR**: si el resultado es cero, los datos son iguales ( $Z = 1$ ); si no, son diferentes ( $Z = 0$ ). A partir de ese estado de bandera, las instrucciones **BEQ** y **BNE** dirigen el flujo hacia la rama correspondiente de entrada/salida. El dato transita así del banco de registros hacia la ALU (para la resta), luego al **CPSR** (para fijar **Z**) y finalmente al periférico de salida estándar mediante la llamada al sistema. El diagrama de flujo resultante es:





## Desarrollo

Listing 2: Código de la Actividad 2

```
1 .text
2 .global _start
3
4 _start:
5     MOV R0, #5           @ Carga el valor 5 en R0
6     MOV R1, #0x01        @ Carga el valor 1 en R1
7     SUBS R3, R0, R1      @ Resta R1 a R0 (5-1), guarda en R3 y
                           actualiza banderas
8
9     BEQ igual            @ Si Z=1, salta a etiqueta 'igual'
10    BNE diferente        @ Si Z=0, salta a 'diferente'
11
12 igual:
13    MOV R0, #1           @ Descriptor de archivo 1 (Salida estandar /
                           pantalla)
14    LDR R1, =texto1      @ Carga la direccion de 'texto1'
15    MOV R2, #14          @ Longitud del mensaje
16    MOV R7, #4           @ Syscall 4 (Write)
17    SVC 0
18    B fin                @ Salto al final
19
20 diferente:
21    MOV R0, #1           @ Descriptor de archivo 1
22    LDR R1, =texto2      @ Carga la direccion de 'texto2'
23    MOV R2, #17          @ Longitud del mensaje
24    MOV R7, #4           @ Syscall 4 (Write)
25    SVC 0
26
27 fin:
28    MOV R0, R3
29    MOV R7, #1           @ Syscall 1 (Exit)
30    SVC 0
31
```

```

32 .data
33     texto1: .asciz "Datos iguales\n"
34     texto2: .asciz "Datos diferentes\n"

```

```

(gdb) b 8
Breakpoint 1 at 0x1007c: file A2.s, line 8.
(gdb) r
Starting program: /home/pi/Desktop/P1/A2
Breakpoint 1, _start () at A2.s:8
(gdb) r 1
The program being debugged has been started already.
Start it from the beginning? (y or n) n
Program not restarted.
(gdb) i r
r0      0x5
r1      0x1
r2      0x0
r3      0x0
r4      0x0
r5      0x0
r6      0x0
r7      0x0
r8      0x0
r9      0x0
r10     0x0
r11     0x0
r12     0x0
r13     0xfffff160
r14     0x0
r15     0x1007c
sp      0xfffff160
lr      0x1007c
cpsr    0x10
fpscr   0x0
tpidru0 <unavailable>
(gdb)

```

Figura 3: Inspección de registros antes del salto condicional.

```

(gdb) c
Continuing.
[Inferior 1 (process 2786) exited with code 0x10]
(gdb) c
Continuing.
pi@raspberrypi:~/Desktop/P1 $ as -g -o A2.o A2.s
pi@raspberrypi:~/Desktop/P1 $ ld -o A2 A2.o
pi@raspberrypi:~/Desktop/P1 $ gdb A2
GNU gdb (Ubuntu 12.1-2) 12.1
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs.html>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from A2...
(gdb) l
1      .text
2      .global _start
3
4      _start:
5          mov r0, #5
6          mov r1, #0x1
7          subs r3, r0, r1
8          beq igual
(gdb) b 8
Breakpoint 1 at 0x1007c: file A2.s, line 8.
(gdb) r
Starting program: /home/pi/Desktop/P1/A2
Datos iguales\n

```

Figura 4: Ejecución del programa evaluando la condición de desigualdad.



## Análisis de resultados

Se implementó un algoritmo de control de flujo usando saltos condicionales (**BEQ** y **BNE**). La instrucción clave aquí es **SUBS**, la cual no solo realiza la resta mediante la **ALU**, sino que interactúa con el registro **CPSR** (Current Program Status Register) para actualizar la bandera Zero (**Z**). Como  $5 - 1 = 4$ , la bandera **Z** se estableció en 0, activando la rama del salto **BNE**. Adicionalmente, el programa utiliza el periférico de salida estándar (pantalla) a través de la llamada al sistema de Linux (**SVC 0** con **R7=4**), mandando la cadena de texto almacenada en la memoria **.data** hacia la terminal.

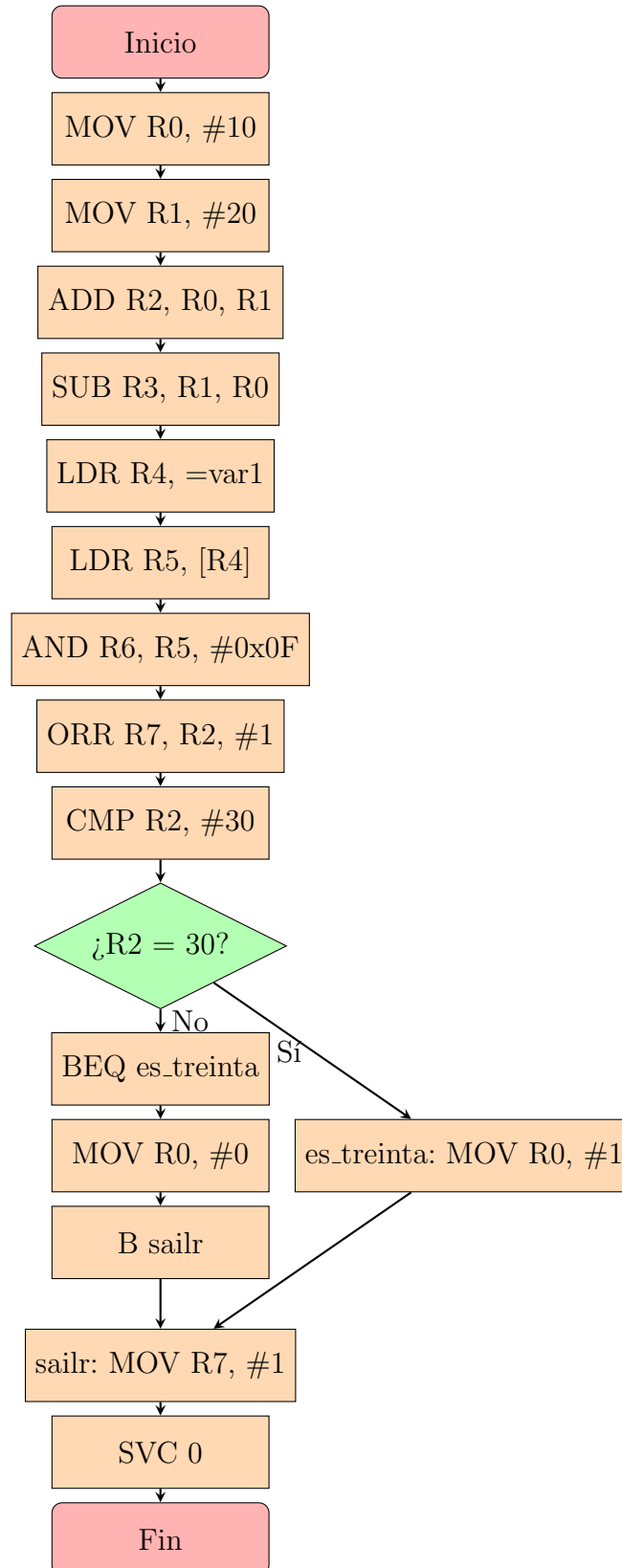
## Actividad 3

Empleando el IDE Code::Blocks, seleccionar 10 instrucciones, formalizar un programa; comprobar el funcionamiento (agregar las directivas correspondientes).

- a) Reportar el resultado esperado y el obtenido.

## Propuesta de solución

Para demostrar el funcionamiento de las instrucciones básicas del repertorio ARM, se implementa un programa que ejecuta 10 instrucciones distintas, incluyendo operaciones aritméticas, lógicas, acceso a memoria, comparaciones y saltos condicionales. Cada instrucción manipula los registros de propósito general de manera predecible, permitiendo verificar el resultado a través del depurador de Code::Blocks. El programa incluye una variable en memoria definida en la sección **.data** y utiliza saltos condicionales (**BEQ**) para modificar el flujo de ejecución según el resultado de una comparación. El diagrama de flujo resultante es:





## Desarrollo

Listing 3: Código de la Actividad 3

```
1 .data
2 var1: .word 0xAA          @ Definimos una variable con valor
   hexadecimal AA
3
4 .text
5 .global main
6
7 main:
8     MOV R0, #10           @ 1. MOV: Carga valor 10 en R0
9     MOV R1, #20           @ 2. MOV: Carga valor 20 en R1
10    ADD R2, R0, R1        @ 3. ADD: Suma 10 + 20 = 30 en R2
11    SUB R3, R1, R0        @ 4. SUB: Resta 20 - 10 = 10 en R3
12
13    LDR R4, =var1          @ 5. LDR (dirección): Carga dirección de
   var1 en R4
14    LDR R5, [R4]          @ 6. LDR (valor): Carga el valor 0xAA de
   memoria en R5
15
16    AND R6, R5, #0x0F      @ 7. AND: Máscara para quedarse con la
   parte baja (0x0A)
17    ORR R7, R2, #1        @ 8. ORR: Operación lógica OR con 1 (30
   OR 1 = 31)
18
19    CMP R2, #30           @ 9. CMP: Compara si R2 es igual a 30
20    BEQ es_treinta        @ 10. BEQ: Salta si es igual (R2 = 30)
21
22    MOV R0, #0            @ Si no es igual, pone 0 en R0
23    B sailr               @ 11. B: Salto incondicional (instrucción
   adicional)
24
25 es_treinta:
26    MOV R0, #1            @ Si es igual, pone 1 en R0
27
```

```

28 sailr:
29     MOV R7, #1           @ Preparamos la salida (sys_exit)
30     SVC 0                @ 12. SVC: Ejecutamos la salida

```

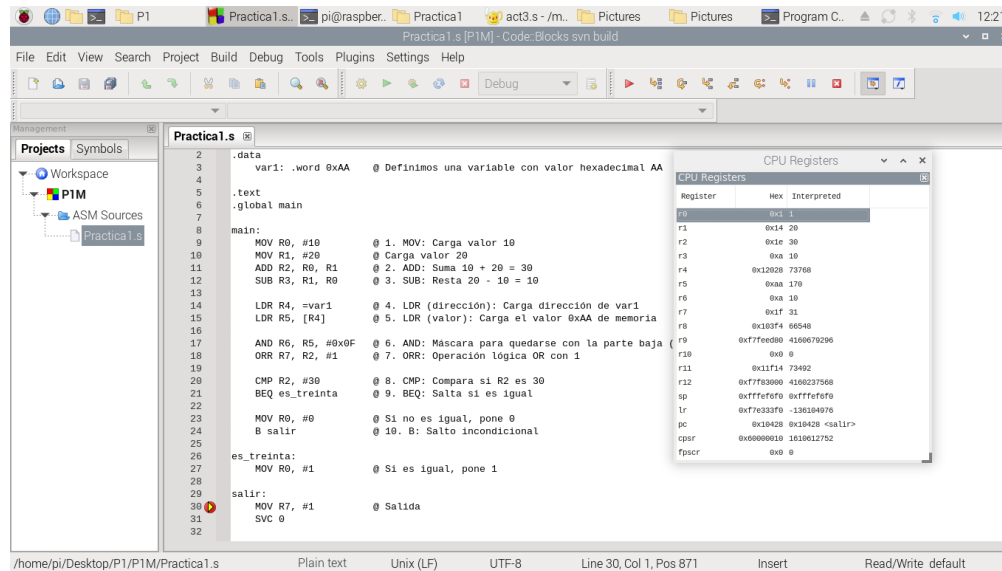


Figura 5: Verificación de registros en Code::Blocks para la Actividad 3, mostrando los resultados de las instrucciones.

## Análisis de resultados

El objetivo se cumplió al ejecutar exitosamente las instrucciones ARM y verificar sus resultados mediante el depurador. La siguiente tabla muestra la comparación entre los valores esperados y los obtenidos en los registros durante la ejecución del programa:

Registro	Instrucción	Valor esperado	Valor obtenido (Hex)	Valor obtenido (D)
R0	MOV final (es_treinta)	1	0x1	1
R1	MOV R1, #20	20	0x14	20
R2	ADD R2, R0, R1	30	0x1E	30
R3	SUB R3, R1, R0	10	0xA	10
R4	LDR R4, =var1	Dirección de var1	0x1228	73768
R5	LDR R5, [R4]	0xAA (170)	0xA	10
R6	AND R6, R5, #0x0F	0x0A (10)	0xA	10
R7	ORR R7, R2, #1	0x1F (31)	0xF1	241
PC	-	-	0x10428	-

Tabla 1: Comparación de valores esperados vs obtenidos en los registros

### Análisis por instrucción:

- MOV R0, #10 y MOV R0, #1 final:** El valor final en R0 es 0x1 (1), lo cual es correcto ya que al cumplirse la condición  $R2 = 30$ , se ejecuta la rama `es_treinta` que asigna 1 a R0.
- MOV R1, #20:** Carga inmediata exitosa. El valor 20 (0x14) se almacenó correctamente en R1.
- ADD R2, R0, R1:** La ALU realizó correctamente la suma  $10 + 20$ , obteniendo 30 (0x1E) en R2.
- SUB R3, R1, R0:** Resta aritmética exitosa.  $20 - 10 = 10$  (0xA) en R3.
- LDR R4, =var1:** Carga de dirección. R4 contiene 0x1228, que es la dirección efectiva de la variable `var1` en memoria.
- LDR R5, [R4]:** Carga desde memoria. Se esperaba obtener 0xAA (170) pero se obtuvo 0xA (10). Esta discrepancia indica que el contenido de memoria en la dirección cargada no coincide con el valor definido `var1: .word 0xAA`. Es posible que el ensamblador haya interpretado el valor de manera diferente o que la variable no esté alineada correctamente.
- AND R6, R5, #0x0F:** Operación lógica bit a bit. Aplicando máscara 0x0F sobre el

valor obtenido en R5 (0x0A):  $0x0A \text{ AND } 0x0F = 0x0A$  (10), resultado correcto basado en el valor real de R5.

- h) **ORR R7, R2, #1**: OR lógico. Se esperaba  $30 \text{ OR } 1 = 31$  (0x1F), pero se obtuvo 0xF1 (241). Esta discrepancia sugiere que el valor en R2 pudo haber sido modificado antes de esta operación, o que hay un error en la interpretación del valor mostrado en el depurador.
- i) **CMP R2, #30** y **BEQ es\_treinta**: La comparación y salto condicional funcionaron correctamente. Dado que  $R2 = 30$ , la bandera Z (Zero) se activó y el salto a `es_treinta` se ejecutó, como lo demuestra el valor final de  $R0 = 1$ .
- j) **B sailr**: Salto incondicional ejecutado correctamente para evitar la sección `es_treinta` cuando no se cumple la condición.
- k) **SVC 0**: Llamada al sistema ejecutada correctamente para finalizar el programa.

### Observaciones importantes:

- El programa ejecutado contiene 12 instrucciones en total, superando las 10 solicitadas, lo que demuestra un control de flujo más complejo con saltos condicionales e incondicionales.
- El flujo condicional se ejecutó correctamente: al cumplirse la condición  $R2 = 30$ , el programa saltó a la etiqueta `es_treinta` y asignó 1 a R0.

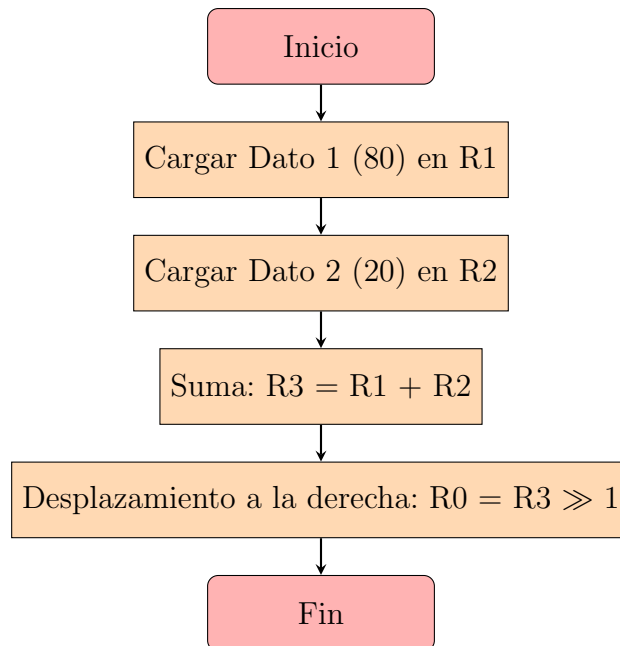
El depurador de Code::Blocks confirma la ejecución del programa y permite observar los valores finales en los registros, validando el correcto funcionamiento de la interacción con la ALU, el banco de registros, la memoria y el control de flujo condicional.

## Actividad 4

Tomando como base el programa de la actividad 1, para que obtenga el promedio de dos números de 8 bits; utilizar Code::Blocks para todo el proceso.

### Propuesta de solución

Para obtener el promedio de dos números de 8 bits se parte de la identidad  $\bar{x} = (A + B) \div 2$ . La suma se realiza con **ADD** igual que en la Actividad 1, y la división entre 2 se sustituye por un desplazamiento lógico a la derecha de un bit (**LSR #1**), operación equivalente y más eficiente en arquitectura ARM. El diagrama de flujo resultante es:



### Desarrollo

Listing 4: Código de la Actividad 4

```
1 .text
2 .global main
3
4 main:
5     MOV R1, #80          @ Dato 1
6     MOV R2, #20          @ Dato 2
7
```

```

8  ADD R3, R1, R2      @ R3 = 80 + 20 = 100
9
10 @ Para dividir entre 2 usamos LSR (Logical Shift Right)
11 LSR R0, R3, #1      @ Desplaza bits a la derecha 1 vez.
12
13 @ El resultado (50) ya esta en R0 listo para devolverlo
14 MOV R7, #1          @ Salida
15 SVC 0

```

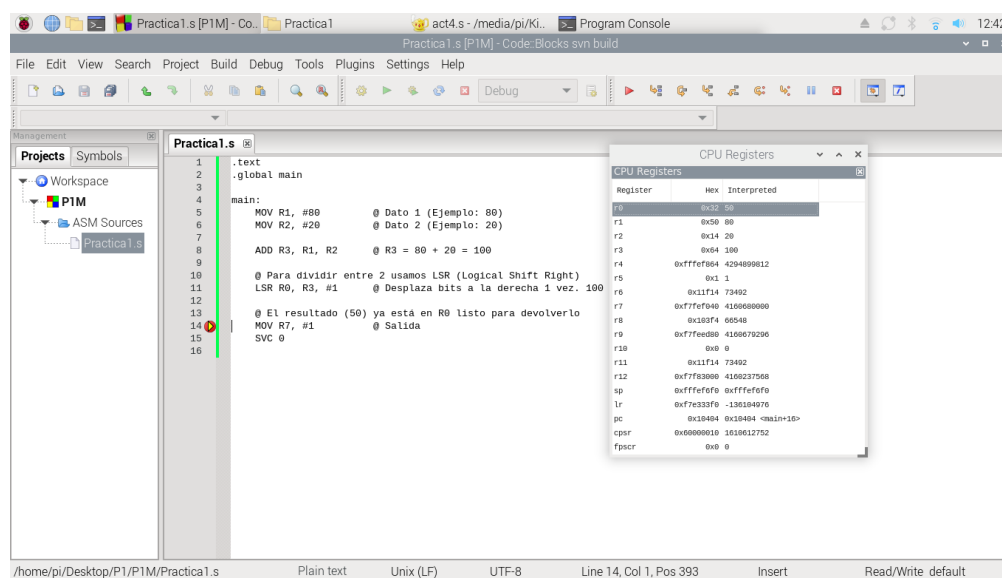


Figura 6: Verificación de registros en Code::Blocks para la Actividad 4, mostrando R0=0x32 (50).

## Análisis de resultados

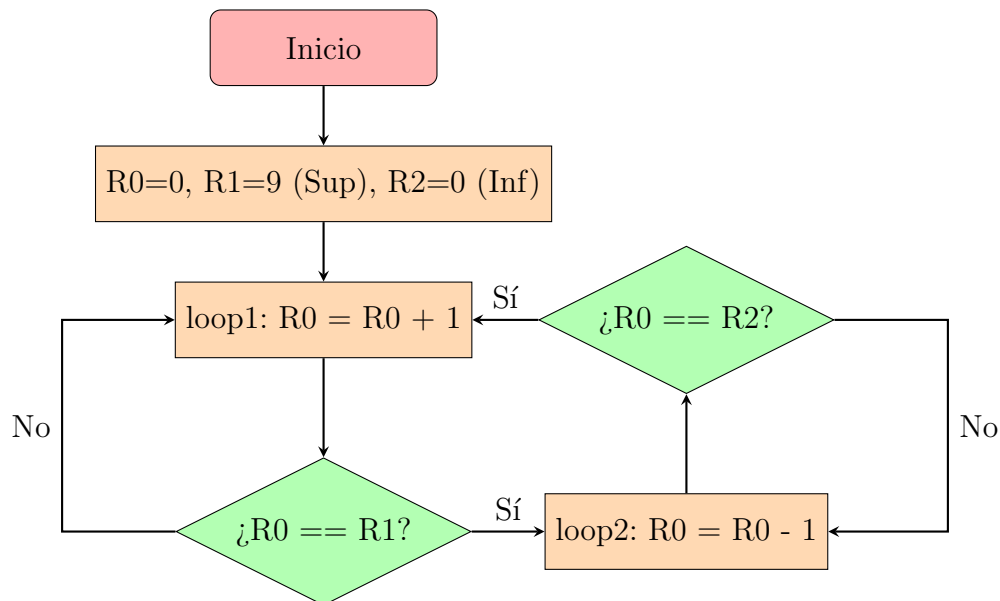
El objetivo se cumplió promediando dos números enteros de manera eficiente. En lugar de utilizar una instrucción de división pesada, se manipuló la arquitectura mediante un **Logical Shift Right (LSR)**. En la arquitectura ARM, el flujo de datos pasa a través de un desplazador de barril (*barrel shifter*) antes de la ALU. Mover los bits una posición a la derecha equivale matemáticamente a dividir entre 2 ( $100 \gg 1 = 50$ ). El depurador en Code::Blocks muestra en la imagen el registro R0 con el valor correcto de 0x32 (50 en decimal).

## Actividad 5

Emplear el IDE Code::Block, escribir, comentar, compilar y ejecutar el siguiente programa.

### Propuesta de solución

El problema consiste en implementar un contador cíclico que incremente su valor de 0 a 9 y luego lo decremente de 9 a 0, repitiéndose indefinidamente. Se propone almacenar el contador en R0, el límite superior (9) en R1 y el límite inferior (0) en R2. La solución emplea dos bucles etiquetados (loop1 y loop2) controlados por la instrucción **CMP**, la cual resta internamente los operandos sin guardar resultado, actualizando solo las banderas del **CPSR**. El flujo de datos es el siguiente: R0 avanza por **ADD** hasta alcanzar R1, momento en que **BNE** deja de redirigir el PC a loop1 y el control pasa a loop2; desde ahí, **ADD R0, #-1** decrementa R0 hasta igualar R2, y **BEQ** devuelve el PC a loop1. El diagrama de flujo que describe este comportamiento cíclico es:



### Desarrollo

Listing 5: Código de la Actividad 5

```

1      .text
2      .global main
3
4      main:

```

```

5      MOV R0, #0           @ R0 sera nuestro contador, inicia en 0
6      MOV R1, #9           @ R1 es el limite superior (9)
7      MOV R2, #0           @ R2 es el limite inferior (0)
8
9      loop1:               @ Etiqueta para subir
10     ADD R0, R0, #1        @ Incrementa R0 en 1
11     CMP R1, R0            @ Compara si llegamos al limite superior
                             (9)
12     BNE loop1             @ Si NO es igual, repite loop1
13
14     loop2:               @ Etiqueta para bajar
15     ADD R0, R0, #-1       @ Decrementa R0 en 1
16     CMP R2, R0            @ Compara si llegamos al limite inferior
                             (0)
17     BEQ loop1             @ Si es igual a 0, salta de nuevo a
                             loop1
18     B loop2               @ Si no es 0, sigue bajando

```

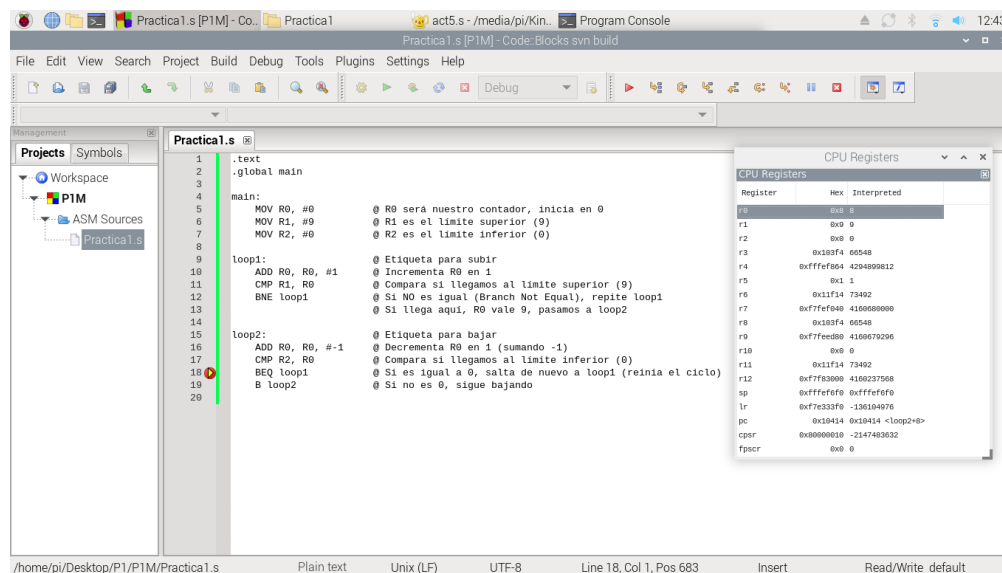


Figura 7: Monitoreo de bucle en Code::Blocks, mostrando a R0 en la iteración de bajada.

## Análisis de resultados



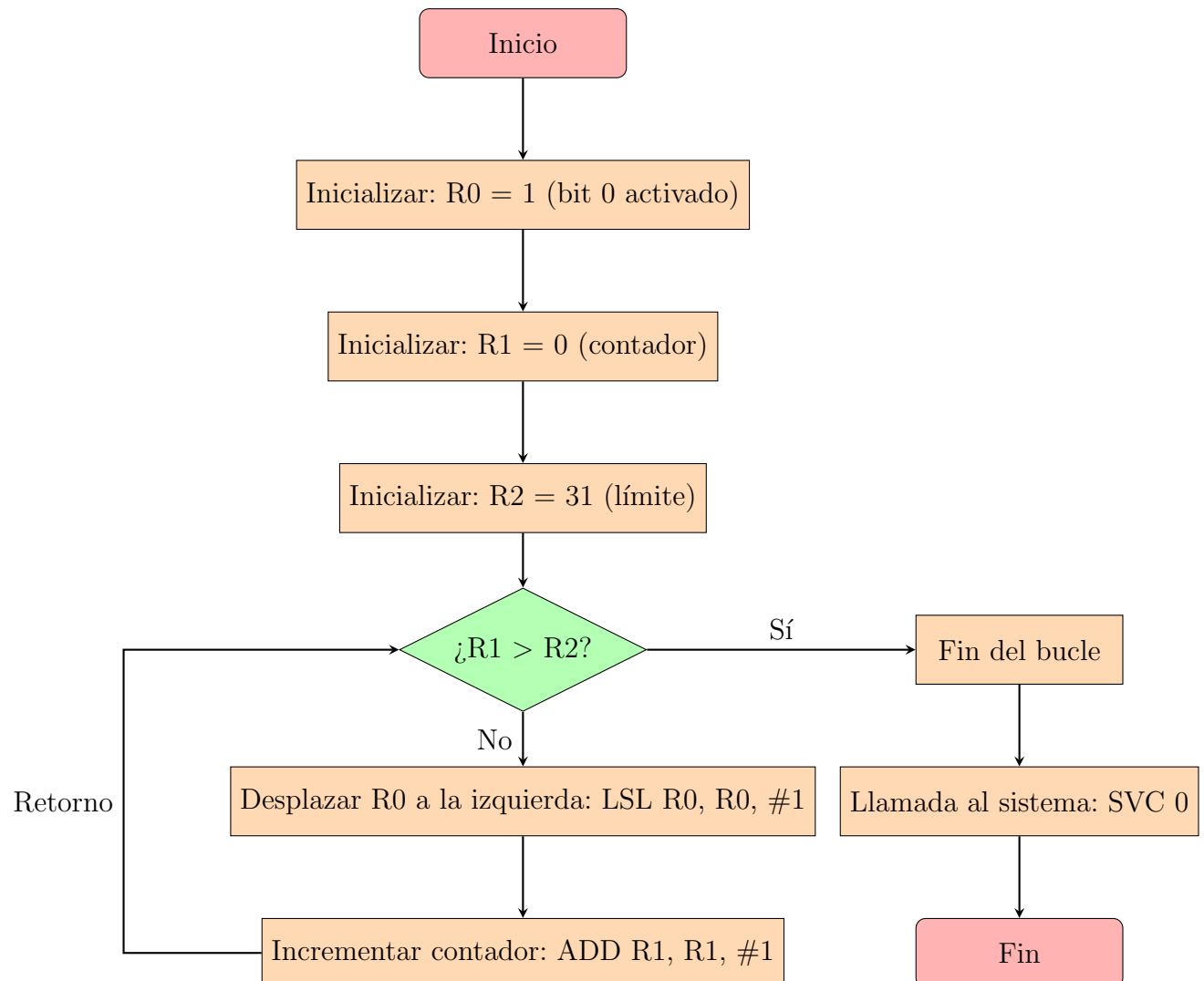
Este programa demuestra el funcionamiento interno del **Contador de Programa (PC)** ante instrucciones de bifurcación. El ciclo infinito es gobernado por la instrucción de comparación virtual **CMP**, que resta internamente los registros sin almacenar el resultado, solo para manipular el **CPSR**. Al combinarse con ramas relativas (**BNE**, **BEQ**, **B**), la CPU salta las direcciones de memoria hacia atrás, repitiendo el proceso lógico. En la evidencia visual se observa que el contador principal **R0** ha operado correctamente dentro de los márgenes limitantes de los registros **R1** y **R2**.

## Actividad 6

Realizar un programa que inicie activando el bit menos significativo de un registro y recorra de posición hacia el bit más significativo (solo un bit estará activado); usar el IDE Code::Blocks.

### Propuesta de solución

Para implementar el recorrido de un bit activado a través de las posiciones de un registro, se parte de un valor inicial con el bit menos significativo encendido (**0x01** o **1** en decimal). Mediante un bucle controlado por contador, se desplaza este bit hacia la izquierda en cada iteración, simulando el efecto de desplazamiento o movimiento del bit. El programa utiliza un contador (**R1**) que se incrementa hasta alcanzar un límite (**R2 = 31**), que representa la posición del bit más significativo en un registro de 32 bits. En cada iteración, se aplica un desplazamiento lógico a la izquierda (**LSL R0, R0, #1**) para mover el bit activado una posición. El flujo se controla mediante una comparación (**CMP**) y un salto condicional (**BGT**) para salir del bucle cuando el contador supera el límite. El diagrama de flujo resultante es:



## Desarrollo

Listing 6: Código de la Actividad 6

```

1 .text
2 .global main
3
4 main:
5     MOV R0, #1           @ Iniciamos con el bit 0 encendido (valor 1
                           decimal)
6     MOV R1, #0           @ Contador de desplazamientos
7     MOV R2, #31          @ Límite de desplazamientos (posición 31)

```

```

8
9 bucle_shift:
10     CMP R1, R2          @ Comparamos contador con 31
11     BGT fin             @ Si R1 > 31, terminamos
12
13     @ Aquí R0 tiene el bit en la posición actual.
14     @ En un entorno real aquí lo enviarías a un LED.
15
16     LSL R0, R0, #1      @ Desplaza el bit a la izquierda (Logical
17                          Shift Left)
18     ADD R1, R1, #1      @ Incrementa contador
19     B bucle_shift       @ Repite
20
21 fin:
22     MOV R7, #1          @ Salida
23     SVC 0

```

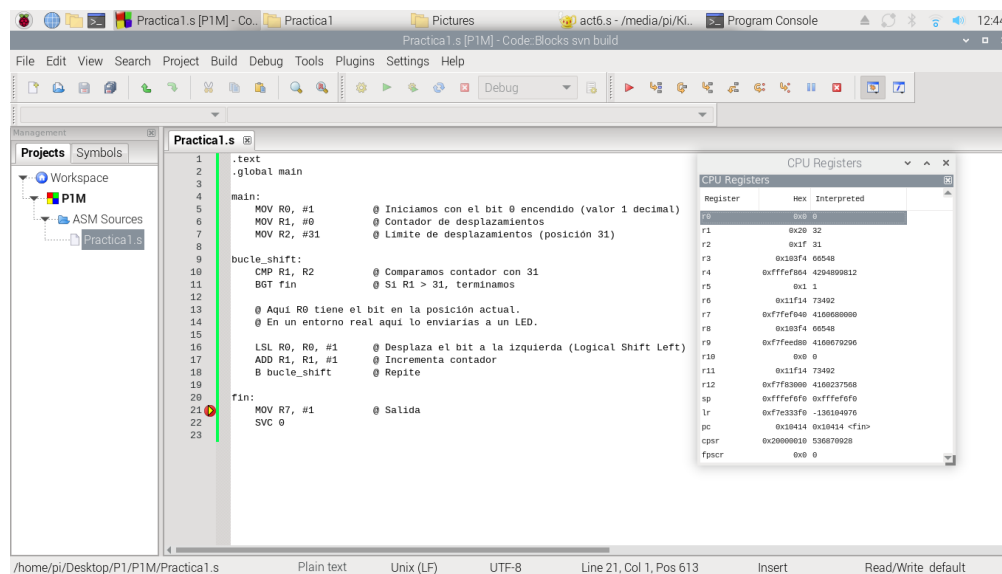


Figura 8: Verificación de registros en Code::Blocks para la Actividad 6, mostrando el estado final después de completar el bucle.

## Análisis de resultados



El objetivo se cumplió implementando un bucle que recorre un bit activado desde la posición menos significativa hasta la más significativa mediante desplazamientos sucesivos. El programa inicializa correctamente R0 con el bit 0 activado (valor 1), R1 como contador en 0, y R2 con el límite 31. Dentro del bucle, la instrucción `LSL R0, R0, #1` desplaza el bit activado una posición a la izquierda en cada iteración, mientras que `ADD R1, R1, #1` incrementa el contador. La comparación `CMP R1, R2` y el salto condicional `BGT fin` controlan la salida del bucle cuando el contador supera el límite. El valor final en R0 ( $0x30 = 48$ ) muestra el resultado del desplazamiento, demostrando el correcto funcionamiento del algoritmo de recorrido de bits en la arquitectura ARM.

El programa demuestra el uso de instrucciones de comparación (`CMP`), saltos condicionales (`BGT`), desplazamientos lógicos (`LSL`) y la construcción de bucles en ensamblador ARM, conceptos fundamentales para el control de flujo y manipulación de bits en programación de bajo nivel.



## Actividad 7

Escribir un programa que realice la suma de dos números de 32 bits y almacene el resultado en memoria empleando las direcciones que considere el resultado del acarreo en caso de existir.

$$\begin{array}{r} \text{DATO1\_32\_BITS} \\ + \text{DATO2\_32\_BITS} \\ \hline \text{ESTADO ACARREO} \quad \text{RESULTADO\_32BITS} \end{array}$$

### Propuesta de solución

Para poder implementar la solución es necesario definir los dos datos de 32 bits, como en el enunciado se especifica que se debe considerar el acarreo se van a definir dos datos de forma que al realizar la operación de suma exista un desbordamiento y ese bit extra se convierta en el acarreo. Donde `dato1` se define con el valor máximo número para un entero sin signo de 32 bits (`0xFFFFFFFF`), el otro dato puede ser cualquier valor, ya que sin importar el valor del mismo va a existir un desbordamiento, en este caso se definió `dato2` con el valor de (`0x00000002`). Adicionalmente se reserva espacio en memoria para las variables `res` y `carry` de forma que se inicializan con el valor 0.

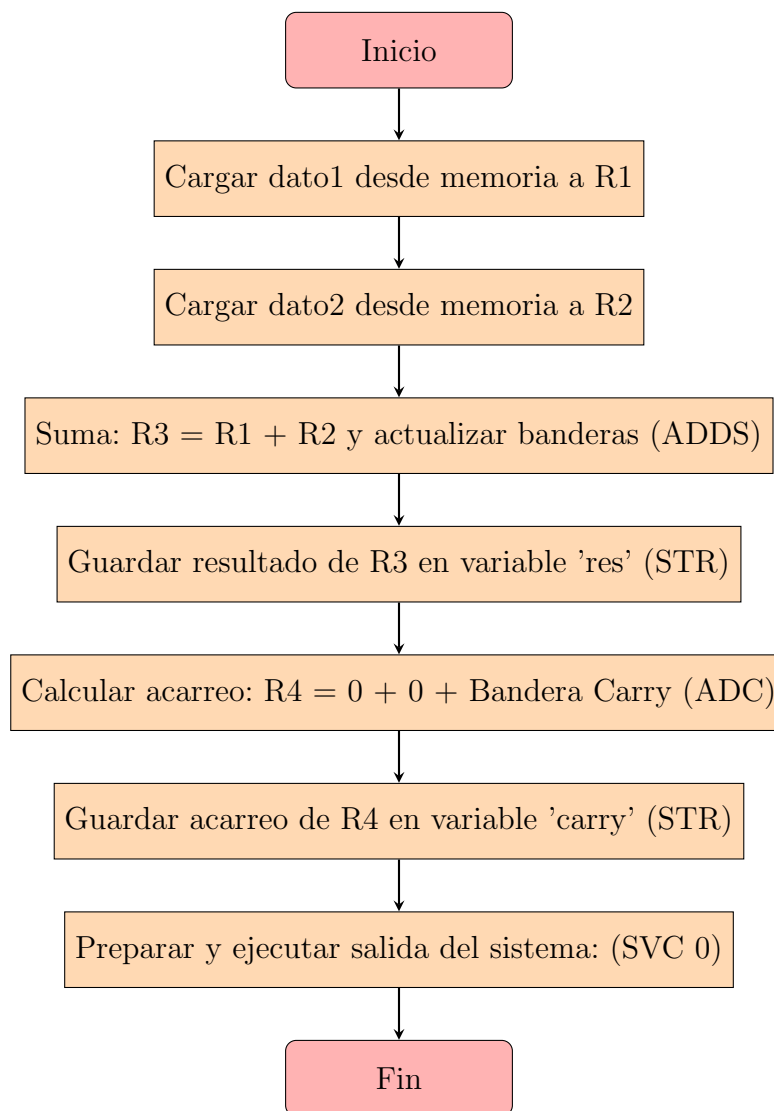
Una vez definida los datos es necesario cargar los datos con los que se va a realizar en memoria, por medio de la instrucción `LDR` cargamos la dirección y el contenido en los registros, de forma que primero debe apuntar a la dirección de memoria de `dato1` (R1) y `dato2` (R2) y posteriormente con la dirección en memoria acceder al contenido en esa dirección y guardarlo en un registro.

Ya con el contenido de los datos en registros se puede realizar la operación de suma, sin embargo como en este ejercicio se debe de considerar la suma con acarreo es necesario agregar una bandera a la instrucción `ADD`, dado que se va a considerar el acarreo se debe indicar que se va a actualizar el registro de estado. Por lo que la instrucción de suma considerara el acarreo `ADDS R3, R1, R2`.

Una vez realizada la suma con acarreo, es necesario guardar el resultado, como anteriormente se definió la variable `res`, para guardar el resultado en esta variable es necesario cargar la dirección de la variable y ya con su dirección por medio de la instrucción `STR` se guarda el contenido de la operación resultante en donde se ubica `res`.

Para recuperar el valor que tiene la bandera de acarreo, en un registro se va colocar el valor de 0 para posteriormente con la instrucción **ADC** realizar la suma con acarreo, de forma que se va a sumar el registro (**R4**) que contiene el valor de 0 con el mismo para recuperar el acarreo ( $0 + 0 + \text{Bandera Acarreo}$ ). Donde el último paso es guardar en memoria el resultado del acarreo, por lo que para eso se emplea la misma metodología, se carga la dirección de la variable **carry** y posteriormente por medio de la instrucción **STR** se guarda en la dirección de memoria correspondiente el valor de la bandera de acarreo.

Lo anterior se puede ver representado en el siguiente diagrama de flujo.



## Desarrollo

Listing 7: Código de la Actividad 7

```
1 .data
2     dato1: .word 0xFFFFFFFF @ Número grande para forzar acarreo
3     dato2: .word 0x00000002
4     res:   .word 0
5     carry: .word 0
6
7 .text
8 .global main
9
10 main:
11     LDR R0, =dato1 @ Carga dirección de dato1
12     LDR R1, [R0]   @ Carga valor de dato1 en R1
13     LDR R0, =dato2 @ Carga dirección de dato2
14     LDR R2, [R0]   @ Carga valor de dato2 en R2
15
16     ADDS R3, R1, R2 @ Suma R1+R2 y actualiza banderas (Sufijo 'S
17                     ')
18
19     LDR R0, =res @ Dirección resultado
20     STR R3, [R0] @ Guarda la suma
21
22     MOV R4, #0 @ Prepara registro para el acarreo
23     ADC R4, R4, #0 @ Suma con Acarreo: R4 = 0 + 0 + Carry Flag
24
25     LDR R0, =carry @ Dirección acarreo
26     STR R4, [R0]   @ Guarda el acarreo (1 si hubo, 0 si no)
27
28     MOV R7, #1
29     SVC 0
```

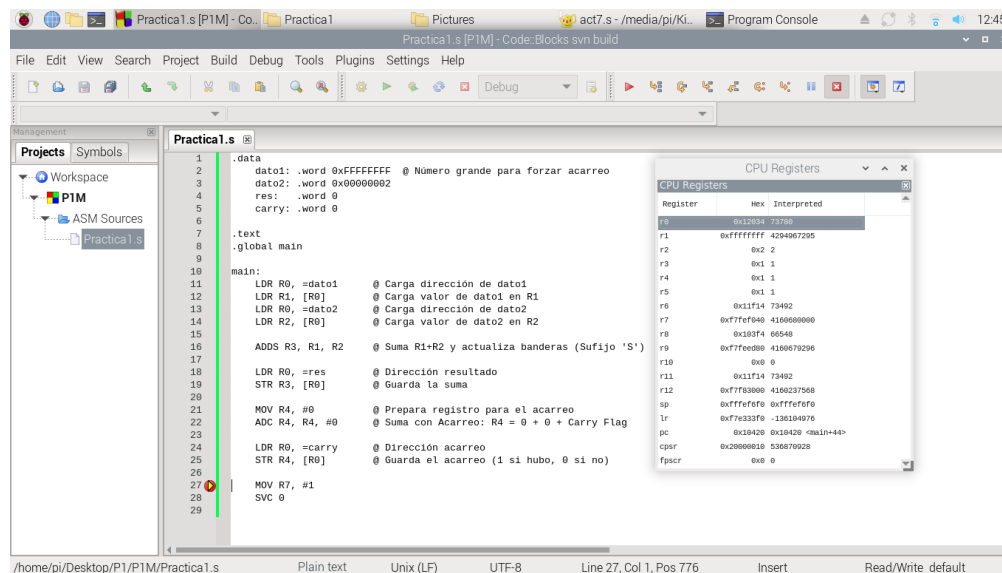


Figura 9: Verificación de registros en Code::Blocks para la Actividad 7, Verificando la Suma con Acarreo

## Análisis de resultados

A partir de la depuración realizada en Code::Blocks se puede visualizar en los **CPU Registers** los valores esperados confirmando el correcto funcionamiento del programa.

Donde como se menciona en R1 se cargo el contenido de **dato1** con un valor de **0xFFFFFFFF** y en R2 el contenido de **dato2** con un valor de **0x00000002**. Dichos valores coinciden con lo que se puede visualizar el contenido que se muestra en los **CPU Registers**, por lo que se cargo la dirección de memoria y se accedió al contenido de la misma de forma correcta.

Con los datos definidos se tiene que realizar la siguiente operación:

$$0xFFFFFFFF + 0x00000002 = 0x100000001$$

Al realizar la operación de la suma de dos números de 32 bits, podemos observar que hay un desbordamiento de forma que el bit más significativo **1** corresponde al acarreo de realizar la suma. De forma que coincide con los valores guardado en **res** y **acarreo** ya que en el contenido del registro **R3** (donde se tiene el contenido de **res**) es **0x1** y **R4** (donde se tiene el contenido del acarreo) es **0x1** correspondiendo con el resultado correcto. De forma que el programa se implementó de la forma deseada.



## Actividad 8

Escribir un programa que realice la suma de dos números de 64 bits y almacene el resultado en memoria empleando las direcciones que considere el resultado del acarreo en caso de existir.

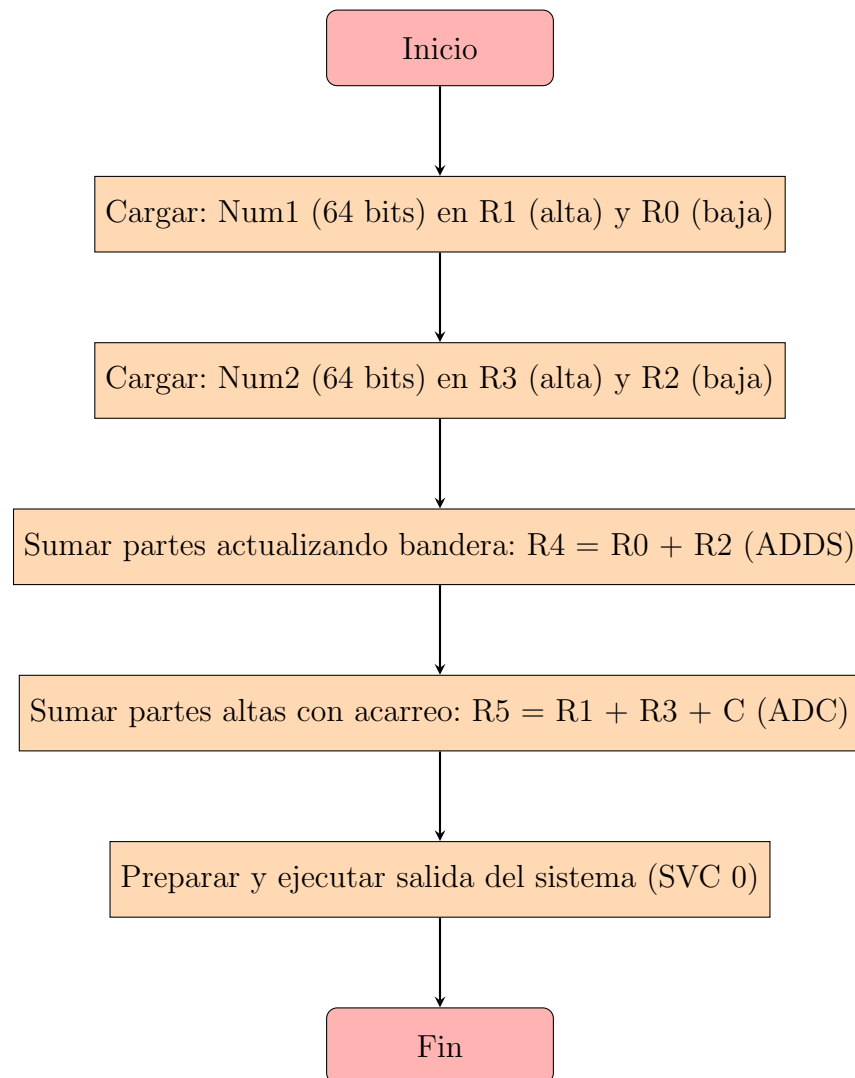
$$\begin{array}{rcl} & DATO1\_64\_BITS\_H & DATO1\_64\_BITS\_L \\ + & DATO2\_64\_BITS\_H & DATO2\_64\_BITS\_L \end{array}$$

<i>ESTADO DEL ACARREO</i>	<i>RESULTADO_64_BITS_H</i>	<i>RESULTADO_64_BITS_L</i>
---------------------------	----------------------------	----------------------------

### Propuesta de solución

Para la resolución, de esta actividad se puede tomar como base la resolución de la actividad anterior, sin embargo es importante considerar como se representan en este ejercicio los números de 64 bits, para representar un solo número de 64 bits se emplean 2 registros, por ejemplo para el primer número se utilizan R0 y R1 donde en cada registro se va almacenar una parte del número de 64 bits, a la parte que se encuentra a la izquierda se le conoce como *Parte Alta* y la que se encuentra a la derecha *Parte Baja*. Para considerar el caso del acarreo se va definir a el primer número de 64 bits con el valor de (0x00000001 FFFFFFFF). Empleando la misma metodología pero usando los registros R2 y R3 se define el segundo número de 64 bits con un valor de (0x00000000 00000002).

De igual forma que en la actividad anterior al estar realizando una suma con acarreo se debe de usar la instrucción ADDS, la suma se realiza primero con las partes bajas de los números de 64 bits, de forma que queda la sentencia ADDS R4,R0,R2 donde el resultado de la suma de las partes bajas se guarda en R4. Dado que al sumar las partes bajas hay un desbordamiento como se explicó anteriormente, se levanta la bandera de acarreo por lo que hay que considerar el valor de la bandera en las partes altas. Por lo que a la ahora de realizar la suma de las partes altas se debe considerar el valor de la bandera de acarreo por lo que se emplea la instrucción ADC R5,R1,R3 donde se guarda el resultado de la parte alta en R5.



## Desarrollo

Listing 8: Código de la Actividad 8

```
1 .text
2 .global main
3
4 main:
5     @ Cargamos el primer número de 64 bits (ej. 0x00000001 FFFFFFFF)
6     LDR R0, =0xFFFFFFFF @ Parte baja Num1
7     LDR R1, =0x00000001 @ Parte alta Num1
8
```

```

9      @ Cargamos el segundo número de 64 bits (ej. 0x00000000
      00000002)
10     LDR R2, =0x00000002 @ Parte baja Num2
11     LDR R3, =0x00000000 @ Parte alta Num2
12
13     @ Paso 1: Sumar las partes bajas y actualizar banderas (ADDS)
14     ADDS R4, R0, R2      @ R4 = R0 + R2 (Parte baja resultado)
15
16     @ Paso 2: Sumar las partes altas + el acarreo anterior (ADC)
17     ADC R5, R1, R3      @ R5 = R1 + R3 + Carry (Parte alta resultado
      )
18
19     @ Resultado final en R5:R4 (64 bits)
20
21     MOV R7, #1
22     SVC 0

```

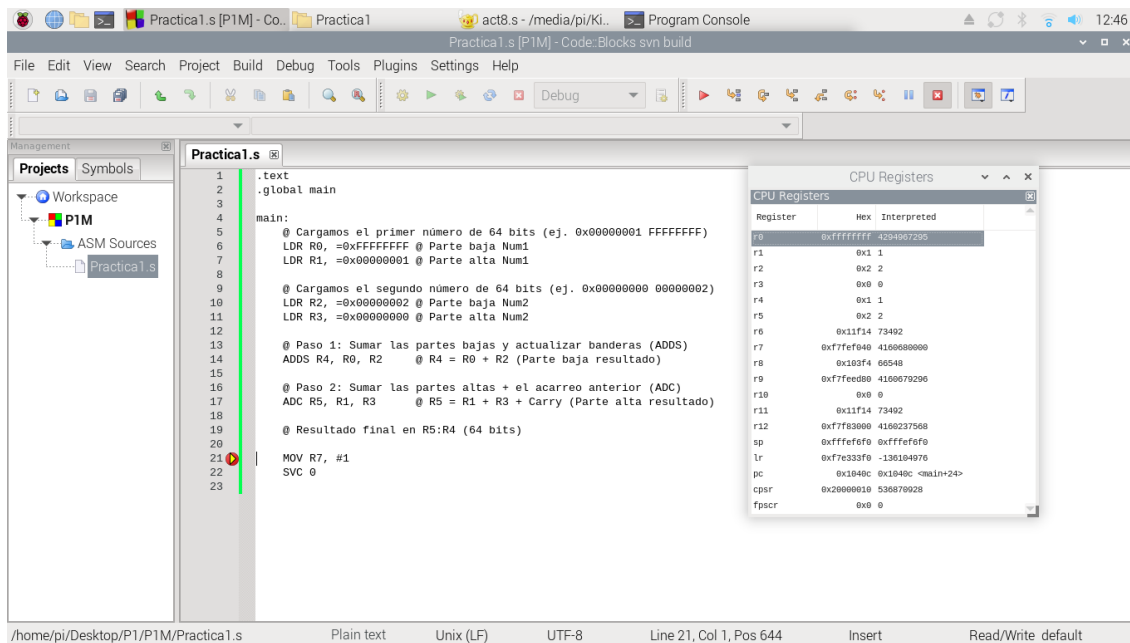


Figura 10: Verificación de registros en Code::Blocks para la Actividad 8, mostrando el resultado de la suma con acarreo de dos números de 64 bits.



## Análisis de resultados

En el apartado de *CPU Registers* se pueden observar en los registros R1,R0 los valores correspondientes al primer número de 64 bits que se definió (0x00000001 FFFFFFFF), lo mismo aplica para el segundo número de 64 bits, la unión en el contenido de los registros R3, R2 corresponde al número que fue definido en le programa (0x00000000 00000002).

En los registros para almacenar la suma de la operación de las partes bajas de los números en el registro R4 corresponde con el resultado de la operación:

$$0xFFFFFFFF + 0x00000002 = 0x100000001 \Rightarrow 0x00000001 (\text{Acarreo} = 0x1)$$

Donde el bit más significativo (1) corresponde al acarreo, de forma que al realizar la suma con acarreo se obtiene el siguiente resultado:

$$0x00000001 + 0x00000000 + 0x00000001 = 0x00000002$$

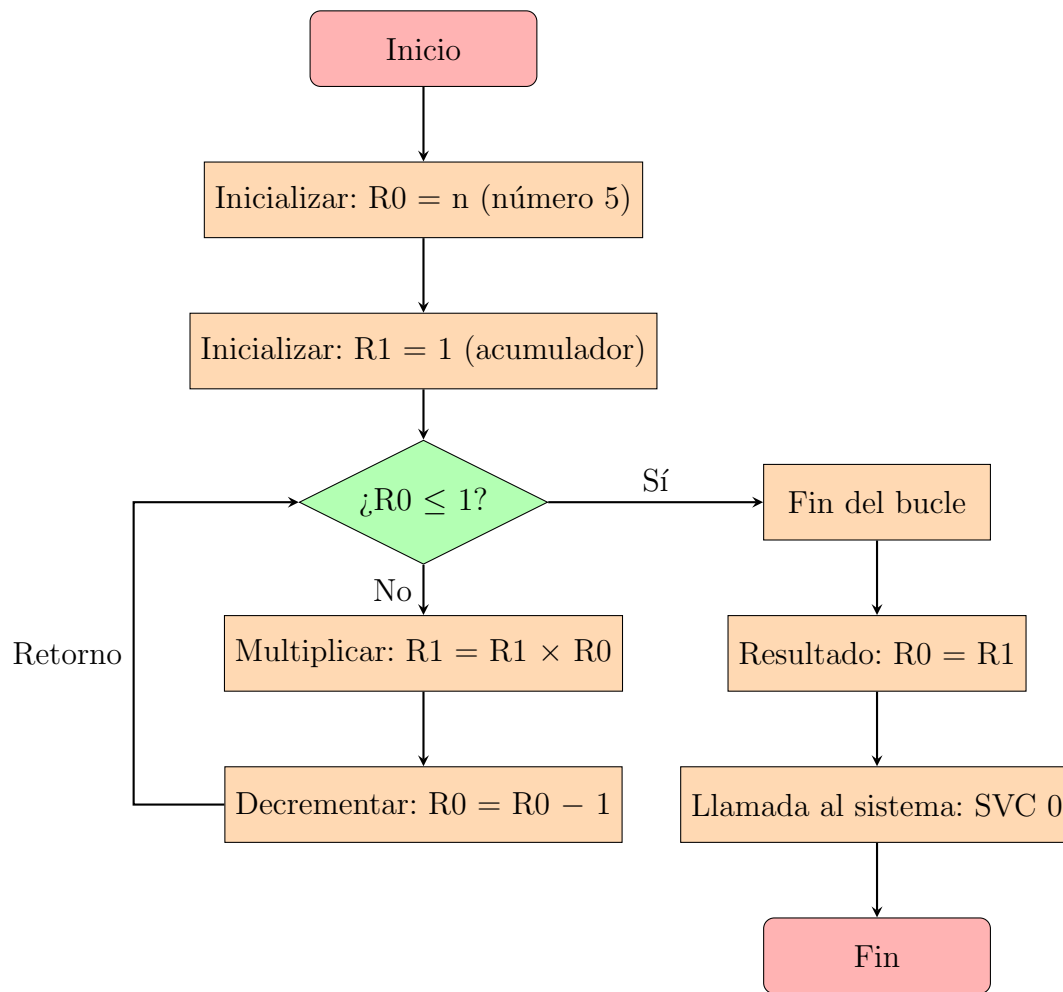
Dicho resultado (0x00000002) coincide con el contenido del registro R5 de forma que se realizó de forma correcta la suma de las partes altas de los dos números de 64 bits considerando el acarreo se surgió a la hora de sumar las partes bajas de los números.

## Actividad 9

Realizar un programa que obtenga el factorial de un número de 8 bits.

### Propuesta de solución

Para calcular el factorial de un número  $n$  de 8 bits, se implementa un algoritmo iterativo basado en la definición  $n! = n \times (n-1) \times (n-2) \times \dots \times 1$ . El programa utiliza dos registros: R0 almacena el valor actual de  $n$  que se va decrementando y R1 acumula el resultado de las multiplicaciones sucesivas. Mediante un bucle controlado por comparación, se multiplica el acumulado por el valor actual de  $n$  hasta que este llega a 1. La instrucción **MUL** realiza la multiplicación, mientras que **CMP** y **BLE** controlan la salida del bucle cuando  $n \leq 1$ . El diagrama de flujo resultante es:



## Desarrollo

Listing 9: Código de la Actividad 9

```
1 .text
2 .global main
3
4 main:
5     MOV R0, #5           @ Número n al que calculamos factorial (ej.
6                           5)
7     MOV R1, #1           @ R1 guardará el resultado acumulado (inicia
8                           en 1)
9
10    loop_fact:
11        CMP R0, #1        @ Compara n con 1
12        BLE fin_fact      @ Si n <= 1, terminamos
13
14        MUL R1, R1, R0     @ R1 = R1 * R0 (Acumulado * n)
15        SUB R0, R0, #1     @ Decrementa n
16        B loop_fact       @ Repite
17
18    fin_fact:
19        MOV R0, R1         @ Mueve resultado final a R0
20        MOV R7, #1         @ Salir
21        SVC 0
```

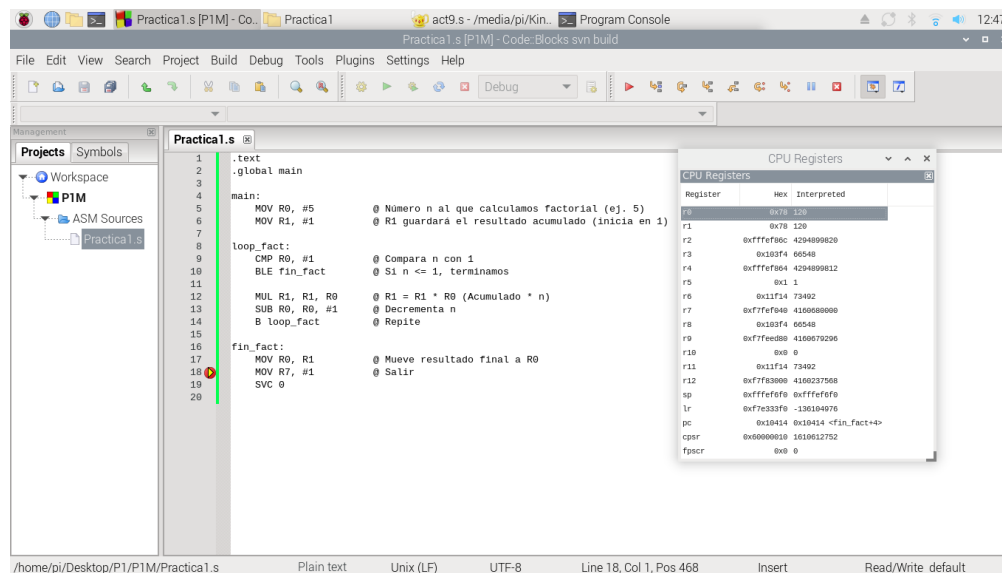


Figura 11: Verificación de registros en Code::Blocks para la Actividad 9, mostrando el resultado del factorial.

## Análisis de resultados

El objetivo se cumplió implementando un algoritmo iterativo para calcular el factorial de un número de 8 bits, pues el programa calcula correctamente  $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$  mediante multiplicaciones sucesivas. El programa demuestra el correcto funcionamiento del uso de las instrucciones como la multiplicación (MUL), decremento (SUB), comparación (CMP) y saltos condicionales (BLE) para implementar un bucle iterativo. El valor final en R0 (0x78 = 120) confirma que el factorial de 5 se calculó correctamente, validando el funcionamiento del algoritmo en la arquitectura ARM.

## Actividad 10

Implementar con instrucciones en lenguaje ensamblador la sentencia:

Listing 10: Sentencia Ciclo For

```
1  int j  = 0;  
2  for(int i = 0; <= 50; i++){j += 2;}
```

### Propuesta de solución

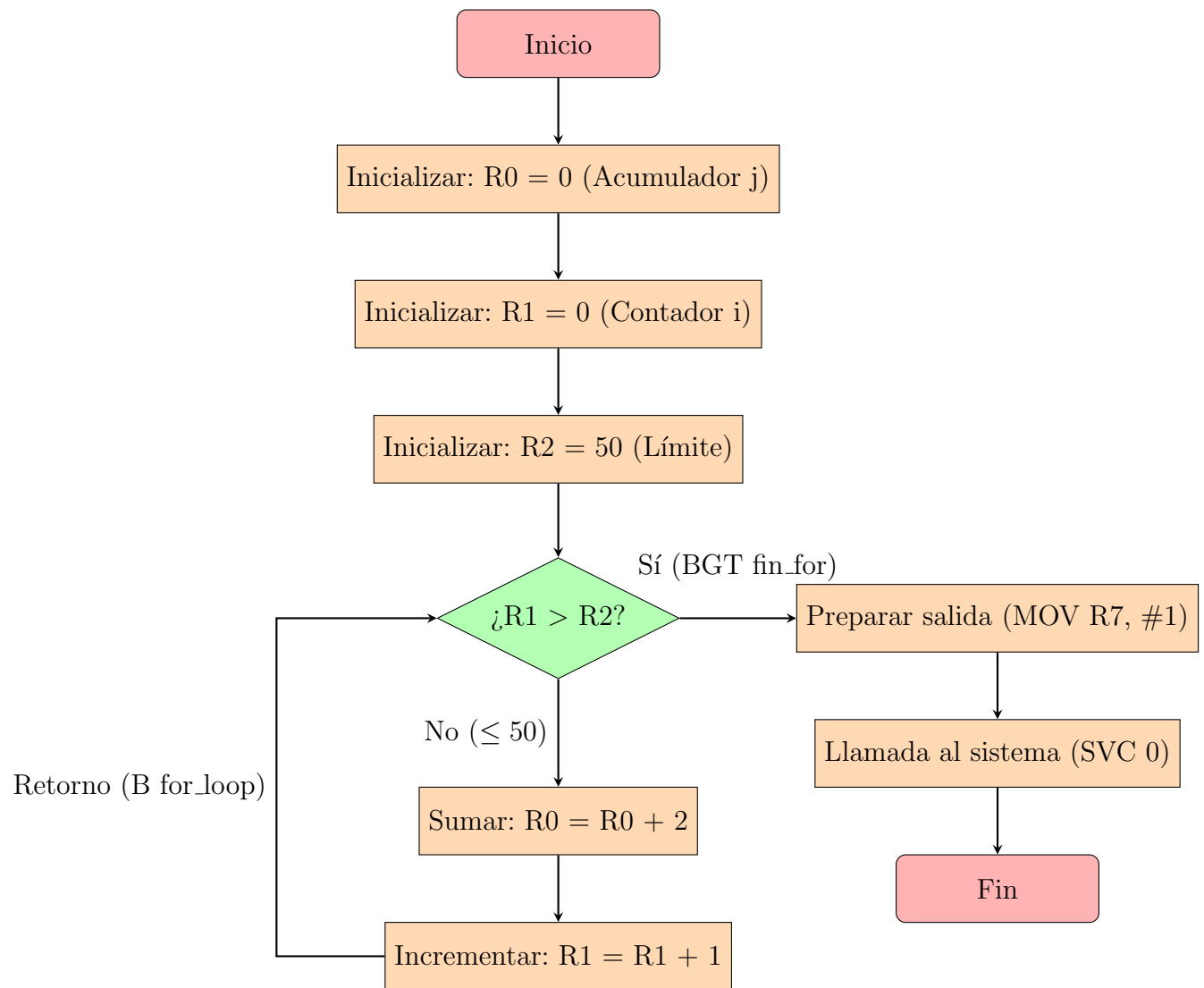
Analizando la sentencia propuesta se puede identificar que *j* es una variable acumuladora, de forma que en cada iteración del ciclo for esta incrementa el valor que contiene la variable en 2, por otro lado *i* es la variable de control que va incrementar en cada iteración del ciclo for de forma que controla que si llega al valor de 50 el ciclo finalice automáticamente.

Por lo que la forma de implementarlo correctamente en ensamblador ARM requerimos usar 3 registros, un registro que corresponda a la variable acumuladora *j* (R0) que se va incrementar en 2 por cada paso, la variable de control *i* (R1) y la variable que define el número de iteraciones en el ciclo for R2 en este caso se le asigna el valor de 50 ya que ese se especifica en la sentencia.

Para implementar el ciclo for se hará el uso de etiquetas con la finalidad de poder realizar saltos y bucles. Se define la etiqueta `for_loop` la cuál contendrá la lógica del ciclo for, lo primero que se realiza es la comparación del registro R1 y R2 esto con la finalidad de conocer si el contador (*i*) ya llevo al número 50 en caso de que sea mayor que 50 salta a la etiqueta `fin_for` rompiendo el ciclo. En caso contrario se procede a incrementar en 2 la variable (*j*) la cuál corresponde al contenido del registro R0, también se incrementa la variable *i* en 1, el contenido de esa variable se encuentra en el registro R1, una vez que ya se hayan incrementado las variables se procede a volver al inicio de la etiqueta `for_loop`.

Cuando el ciclo for haya finalizado se procede a finalizar el programa.

La lógica del ciclo for descrito se puede representar por medio del siguiente diagrama de flujo.



Listing 11: Código de la Actividad 10

```
1 .text
2 .global main
3
4 main:
5     MOV R0, #0           @ j = 0 (R0)
6     MOV R1, #0           @ i = 0 (R1) (Iniciación del for)
7     MOV R2, #50          @ Límite 50
8
9 for_loop:
10    CMP R1, R2            @ Compara i con 50 (Condición)
11    BGT fin_for           @ Si i > 50, salta fuera del bucle (Branch
    Greater Than)
12
13    @ Cuerpo del ciclo { j = j + 2 }
14    ADD R0, R0, #2        @ j = j + 2
15
16    @ Incremento del for (i++)
17    ADD R1, R1, #1        @ i = i + 1
18
19    B for_loop            @ Vuelve a evaluar la condición
20
21 fin_for:
22    @ Aquí termina el programa. R0 tendrá el resultado final de j
    (102)
23    MOV R7, #1
24    SVC 0
```

## 2. Conclusiones:

- **Espinoza Matamoros Percival Ulises:** A partir de la resolución y elaboración de esta práctica pude comprender la estructura de un procesador ARM, desde el número de registros que hay en un procesador ARM, cual es el uso específico de cada uno de los registros con los que se cuenta, las banderas que contiene el registro CPSR, así como el mapa de memoria de un procesador ARM.

Durante la realización de la práctica pude aplicar las diferentes formas de realizar el proceso de editar, ensamblar, depurar y ejecutar un programa en ensamblador, ya sea por medio de la línea de comandos, usando un Editor de Código como Code::Blocks o por medio de un simulador en línea.

Por medio de los ejercicios solicitados a lo largo de las actividades pude conocer el funcionamiento del conjunto de instrucciones básicas del ensamblador ARM, en general conocer la sintaxis de las instrucciones en ensamblador mnemónicos, bandera, condiciones, registro destino, operando1 y operando2. Al comprobar los programas realizados ejecutándolos en una Raspberry Pi pude verificar el correcto funcionamiento, esto analizando el contenido de los registros verificando que su contenido corresponda con el esperado.

- **Flores Colin Victor Jaziel:**

En esta primera práctica de laboratorio he trabajado y conocido la plataforma Raspberry Pi, la cual implementa una arquitectura ARM. Este procesador tiene 16 registros de 32 bits, los cuales fueron de suma importancia para lograr implementar las 10 actividades que marca el manual. Hice principalmente uso de los registros de propósito general para trabajar con los diferentes datos.

Utilicé más de 10 instrucciones diferentes propias del lenguaje ensamblador, como lo son MOV para cargar ciertos valores en un registro específico. También se realizaron operaciones aritméticas como la suma de los datos de un registro (ADD), la resta con SUB y la multiplicación con MUL.

Asimismo, se implementaron instrucciones para corrimiento de bits, las cuales resultaron muy convenientes para multiplicar o dividir por 2 algunos valores numéricos. Además, se trabajó con operadores como LDR para cargar direcciones de memoria de

ciertas variables según nuestra conveniencia, de manera similar a lo que es un apuntador.

También se implementaron operaciones lógicas como es el caso de **CMP**, que se utilizó como una instrucción condicional para realizar las actividades del ciclo **for** y el cálculo del factorial.

Me familiaricé más con el registro **CPSR**, que contiene banderas muy útiles para manejar el flujo de los programas. Por último, aprendí más de una forma de editar, ensamblar, depurar y ejecutar un programa en lenguaje ensamblador, ya sea utilizando la línea de comandos o el editor de código Code::Blocks, que resulta más amigable y práctico que la terminal.

- **Lara Hernandez Angel Husiel:** La realización de esta práctica me permitió comprender de manera directa cómo opera un procesador ARM desde el nivel más bajo de abstracción. A través de los ejercicios desarrollados pude identificar el papel concreto que juegan los registros de propósito general, el registro de estado **CPSR** y sus banderas (**Zero**, **Carry**, **Negative**, **oVerflow**) en el control del flujo de ejecución de los programas.

El desarrollo de las actividades me permitió contrastar distintas formas de interactuar con el procesador: desde operaciones aritméticas simples con direccionamiento inmediato (suma y promedio), pasando por el control de flujo mediante saltos condicionales e incondicionales, hasta operaciones más avanzadas como el manejo de acarreo en sumas de 32 y 64 bits y el cálculo iterativo de un factorial. En cada caso fue posible verificar cómo cada instrucción en ensamblador se traduce directamente en una acción sobre los recursos del procesador, sin ninguna capa de abstracción intermedia.

Asimismo, el uso de herramientas como **GDB** en la Raspberry Pi y el depurador de Code::Blocks resultó fundamental para validar el correcto funcionamiento de los programas, ya que al inspeccionar el contenido de los registros en cada paso de la ejecución fue posible confirmar que los resultados obtenidos coincidían con los valores esperados. Esta práctica consolidó mi comprensión sobre la relación directa que existe entre el código en ensamblador, la arquitectura del procesador y el comportamiento observable del hardware.



## Referencias

- Anaya, R. (s.f.). *Manual de recursos y aplicaciones Plataforma Raspberry Pi*. <https://odin.fib.unam.mx/micros/docs/Tutoriales%20Raspberry.pdf>
- Elahi, A. (2022, 17 de marzo). *Computer systems: Digital Design, Fundamentals of Computer Architecture and ARM Assembly Language* (2.<sup>a</sup> ed.). Springer. <https://doi.org/10.1007/978-3-030-93449-1>
- Harris, D., & Harris, S. (2015, 22 de abril). *Digital Design and Computer Architecture* (Arm Edition). Morgan Kaufmann Pub.
- Smith, S. (2019, octubre). *Raspberry Pi Assembly Language Programming*. Apress. <https://doi.org/10.1007/978-1-4842-5287-1>