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:

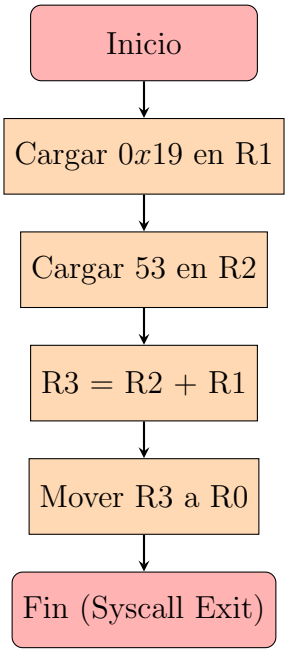
Aprender la estructura de los procesadores con arquitectura ARM, utilizar la plataforma Raspberry Pi, los entornos para programar; desarrollar algoritmos con las instrucciones en lenguaje ensamblador, controlar directamente los recursos del microprocesador; editar, compilar, ensamblar, simular y ejecutar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

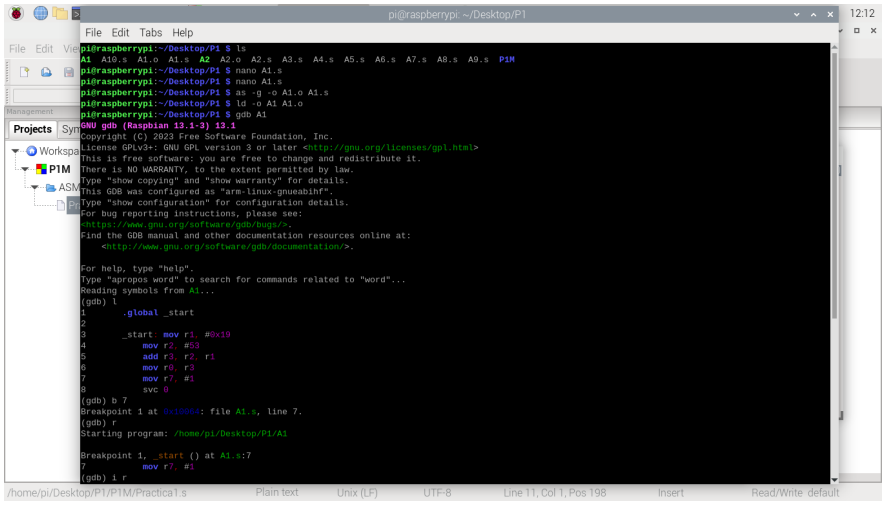


Figura 1: Proceso de ensamblado, enlazado e inicio de GDB en Raspberry Pi.

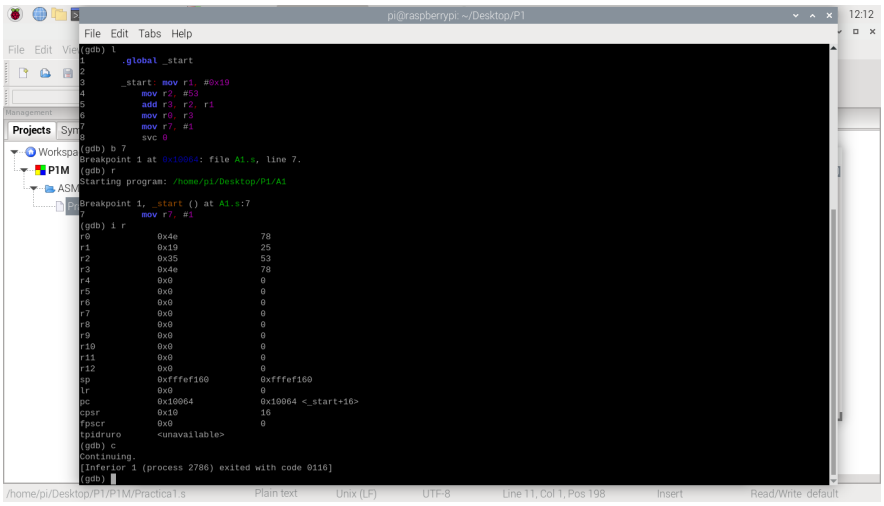


Figura 2: Inspección de registros en GDB mostrando R0 y R3 con el valor de 78 (0x4E).

Análisis de resultados

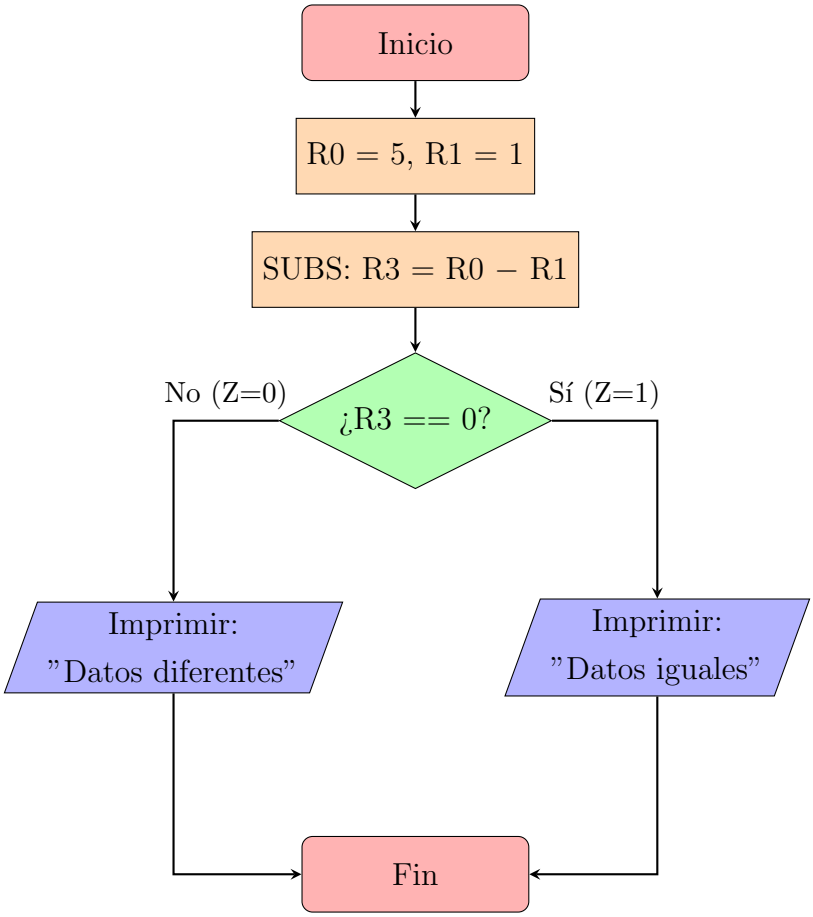
El programa utiliza el **modo de direccionamiento inmediato** para cargar constantes estáticas en los registros internos del procesador (`MOV R1, #0x19` y `MOV R2, #53`). El flujo interno de los datos envía el contenido de los registros R1 y R2 a la Unidad Aritmético Lógica (ALU) a través de la instrucción `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

```
.text
.global _start

_start:
    MOV R0, #5          @ Carga el valor 5 en R0
    MOV R1, #0x01       @ Carga el valor 1 en R1
    SUBS R3, R0, R1     @ Resta R1 a R0 (5-1), guarda en R3 y
       actualiza banderas

    BEQ igual           @ Si Z=1, salta a etiqueta 'igual'
    BNE diferente       @ Si Z=0, salta a 'diferente'

igual:
    MOV R0, #1          @ Descriptor de archivo 1 (Salida estandar /
        pantalla)
    LDR R1, =texto1     @ Carga la direccion de 'texto1'
    MOV R2, #14         @ Longitud del mensaje
    MOV R7, #4          @ Syscall 4 (Write)
    SVC 0
    B fin               @ Salto al final

diferente:
    MOV R0, #1          @ Descriptor de archivo 1
    LDR R1, =texto2     @ Carga la direccion de 'texto2'
    MOV R2, #17         @ Longitud del mensaje
    MOV R7, #4          @ Syscall 4 (Write)
    SVC 0

fin:
    MOV R0, R3
    MOV R7, #1          @ Syscall 1 (Exit)
    SVC 0

```

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

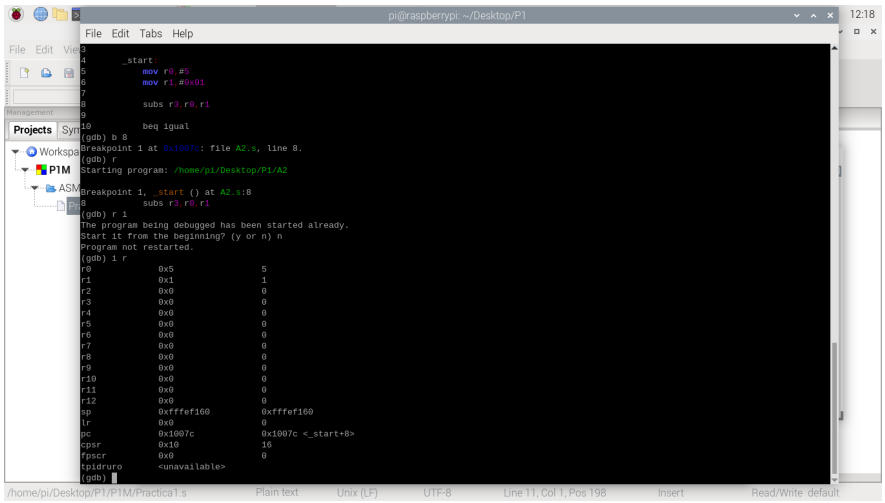


Figura 3: Inspección de registros antes del salto condicional.

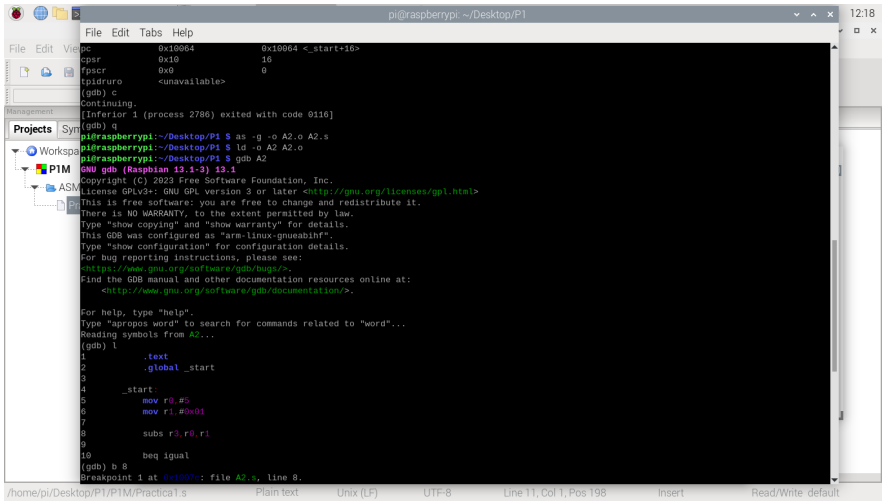


Figura 4: Ejecución del programa evaluando la condición de desigualdad.

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.

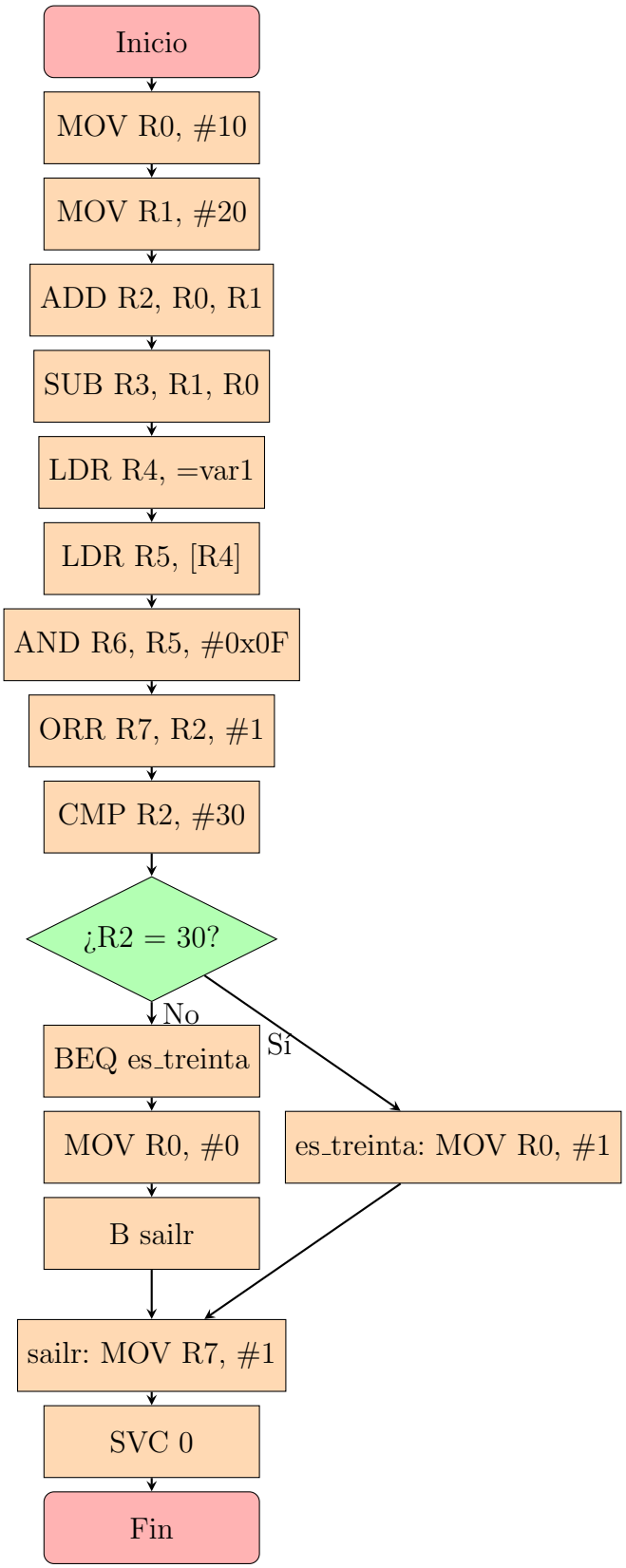
Actividad 3

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:


Inicio
MOV R0, #10
MOV R1, #20
ADD R2, R0, R1
SUB R3, R1, R0
LDR R4, =var1
LDR R5, [R4]
AND R6, R5, #0x0F
ORR R7, R2, #1
CMP R2, #30
¿R2 = 30?
No
Sí
BEQ es_treinta
MOV R0, #0
B sailr
es_treinta: MOV R0, #1
sailr: MOV R7, #1
SVC 0
Fin

Desarrollo

Listing 3: Código de la Actividad 3

```
.data
var1: .word 0xAA            @ Definimos una variable con valor
   hexadecimal AA

.text
.global main

main:
    MOV R0, #10             @ 1. MOV: Carga valor 10 en R0
    MOV R1, #20             @ 2. MOV: Carga valor 20 en R1
    ADD R2, R0, R1          @ 3. ADD: Suma 10 + 20 = 30 en R2
    SUB R3, R1, R0          @ 4. SUB: Resta 20 - 10 = 10 en R3

    LDR R4, =var1           @ 5. LDR (dirección): Carga dirección de
        var1 en R4
    LDR R5, [R4]            @ 6. LDR (valor): Carga el valor 0xAA de
        memoria en R5

    AND R6, R5, #0x0F       @ 7. AND: Máscara para quedarse con la
       parte baja (0x0A)
    ORR R7, R2, #1          @ 8. ORR: Operación lógica OR con 1 (30
       OR 1 = 31)

    CMP R2, #30             @ 9. CMP: Compara si R2 es igual a 30
    BEQ es_treinta          @ 10. BEQ: Salta si es igual (R2 = 30)

    MOV R0, #0              @ Si no es igual, pone 0 en R0
    B sailr                 @ 11. B: Salto incondicional (instrucció
       n adicional)

es_treinta:
    MOV R0, #1              @ Si es igual, pone 1 en R0

```

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

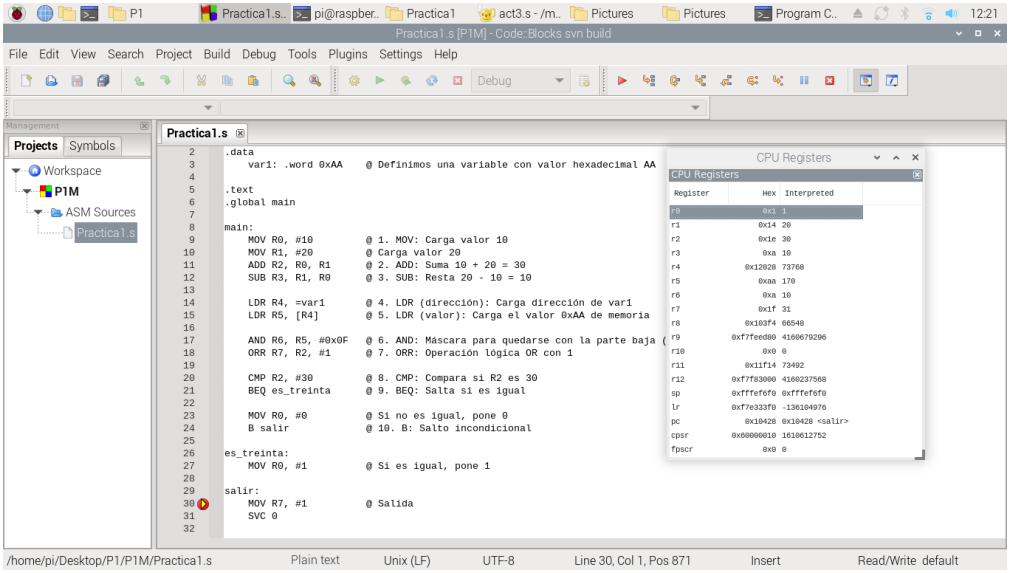


Figura 5: Verificación de registros en Code::Blocks para la Actividad 3, mostrando los resultados de las instrucciones.

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:

a) **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.

b) **MOV R1, #20**: Carga inmediata exitosa. El valor 20 (0x14) se almacenó correctamente en R1.

c) **ADD R2, R0, R1**: La ALU realizó correctamente la suma 10 + 20, obteniendo 30 (0x1E) en R2.

d) **SUB R3, R1, R0**: Resta aritmética exitosa. 20 - 10 = 10 (0xA) en R3.

e) **LDR R4, =var1**: Carga de dirección. R4 contiene 0x1228, que es la dirección efectiva de la variable `var1` en memoria.

f) **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.

g) **AND R6, R5, #0x0F**: Operación lógica bit a bit. Aplicando máscara 0x0F sobre el

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

h) **ORR R7, R2, #1**: OR lógico. Se esperaba 30 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.

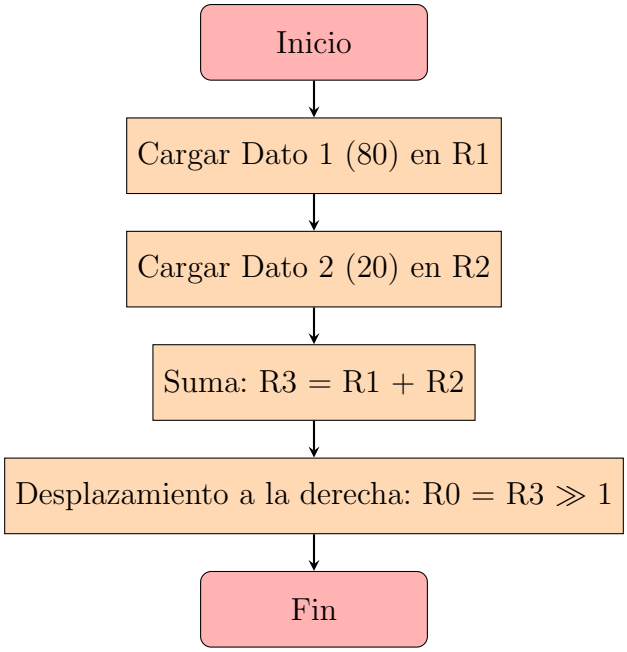
El depurador de Code::Blocks confirma la ejecución del programa y permite observar los valores finales en los registros, validando el correcto funcionamiento de la interacción con la ALU, el banco de registros, la memoria y el control de flujo condicional.

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 $\overline{x} = (A + B) \div 2$. 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

```
.text
.global main

main:
    MOV R1, #80          @ Dato 1
    MOV R2, #20          @ Dato 2

```

```
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
```

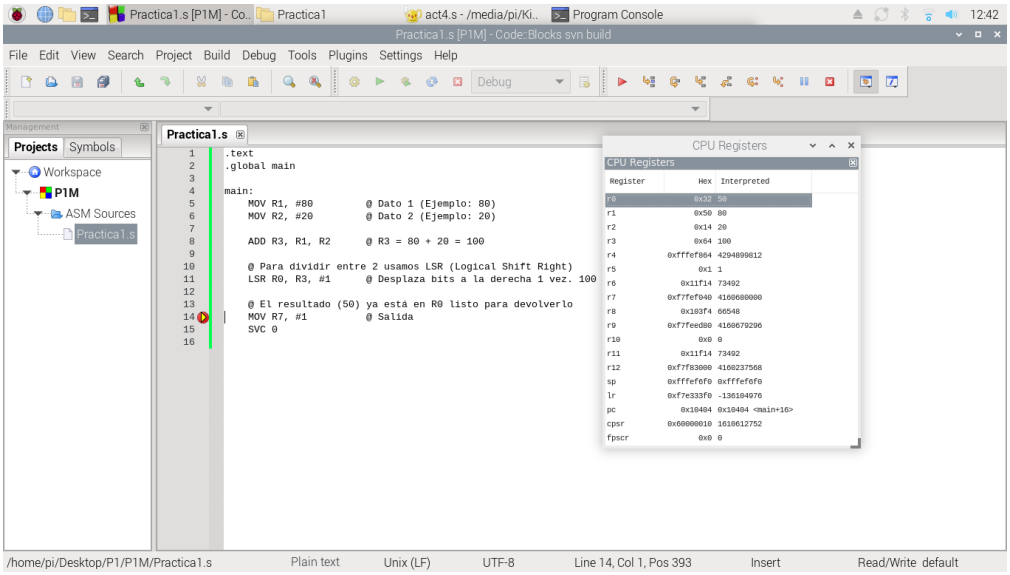


Figura 6: Verificación de registros en Code::Blocks para la Actividad 4, mostrando R0=0x32 (50).

Análisis de resultados

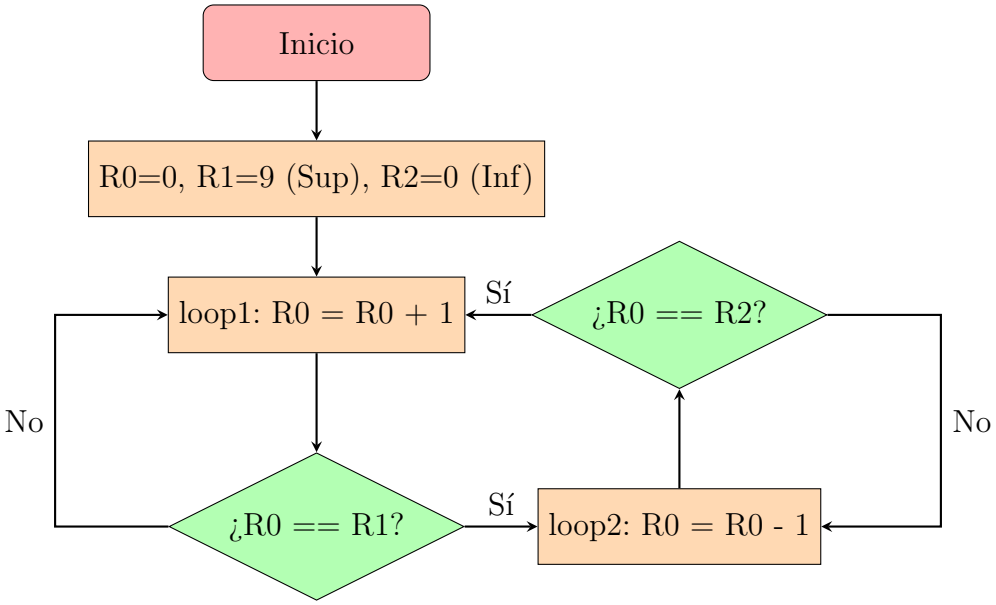
El objetivo se cumplió promediando dos números enteros de manera eficiente. En lugar de utilizar una instrucción de división pesada, se manipuló la arquitectura mediante un **Logical Shift Right (LSR)**. En la arquitectura ARM, el flujo de datos pasa a través de un desplazador de barril (*barrel shifter*) 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
```

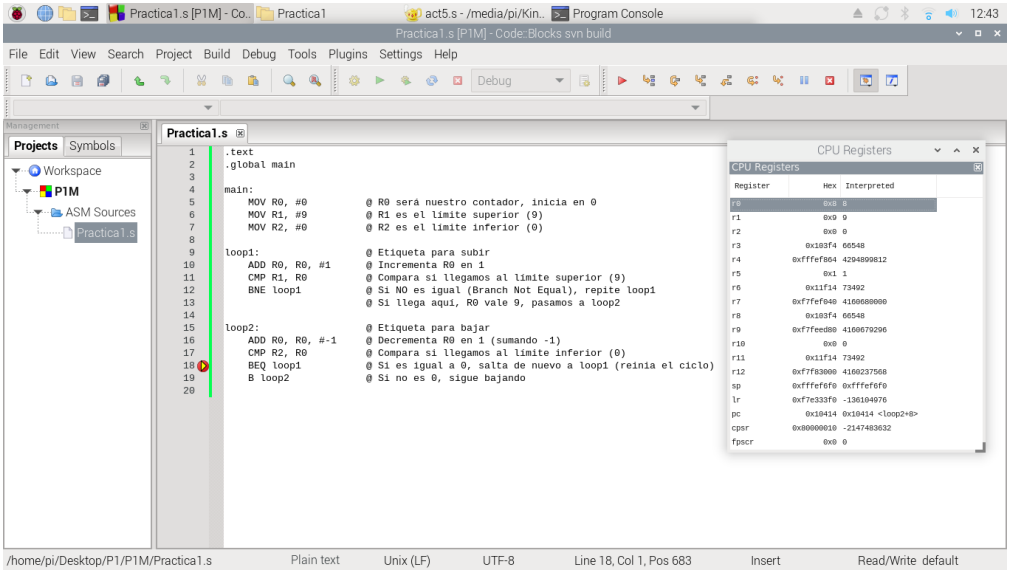


Figura 7: Monitoreo de bucle en Code::Blocks, mostrando a R0 en la iteración de bajada.

Análisis de resultados

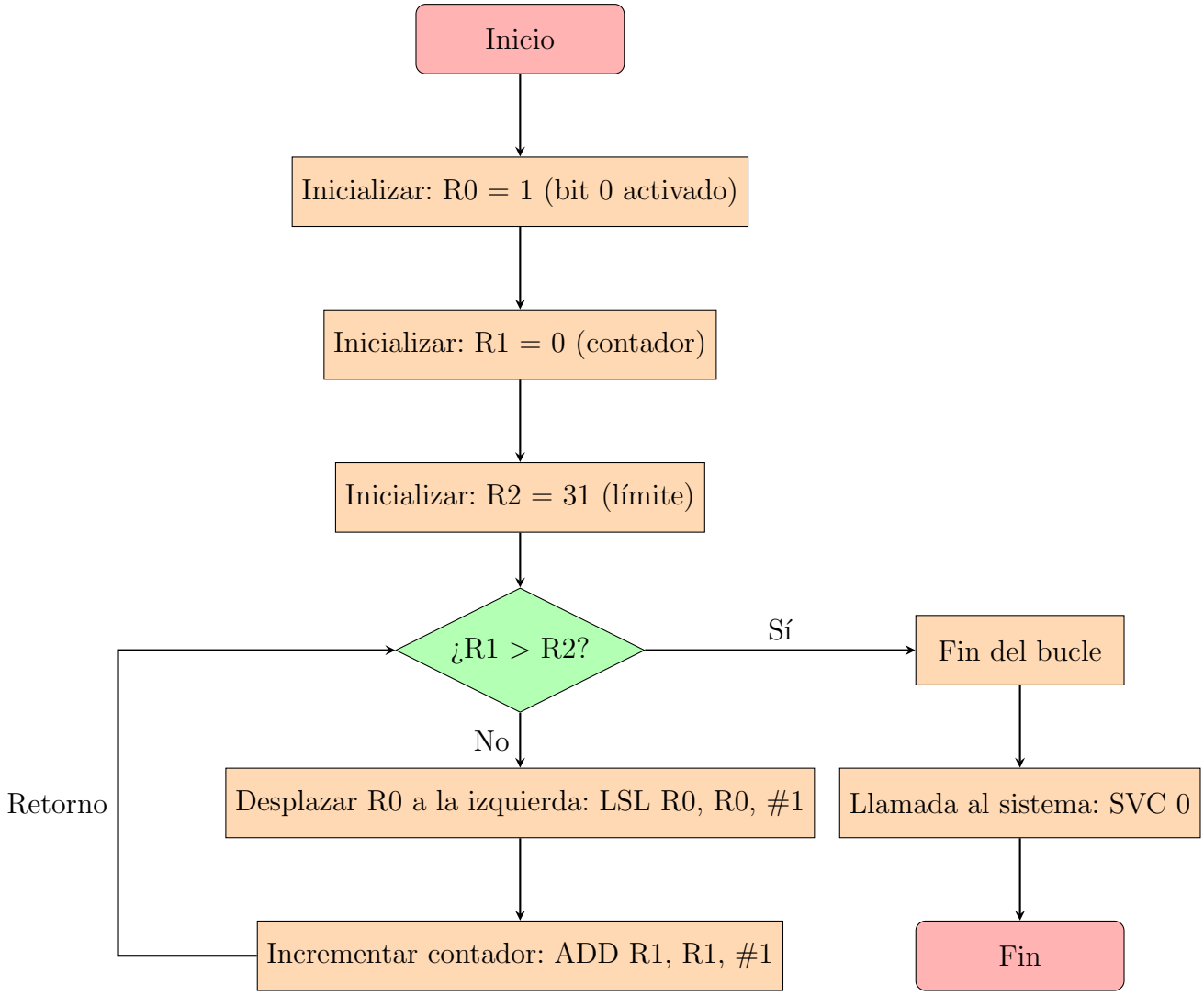
Este programa demuestra el funcionamiento interno del **Contador de Programa (PC)** ante instrucciones de bifurcación. El ciclo infinito es gobernado por la instrucción de comparación virtual `CMP`, que resta internamente los registros sin almacenar el resultado, solo para manipular el `CPSR`. Al combinarse 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

```
.text
.global main

main:
    MOV R0, #1          @ Iniciamos con el bit 0 encendido (valor 1
        decimal)
    MOV R1, #0          @ Contador de desplazamientos
    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
           Shift Left)
17     ADD R1, R1, #1      @ Incrementa contador
18     B bucle_shift       @ Repite
19
20 fin:
21     MOV R7, #1          @ Salida
22     SVC 0
```

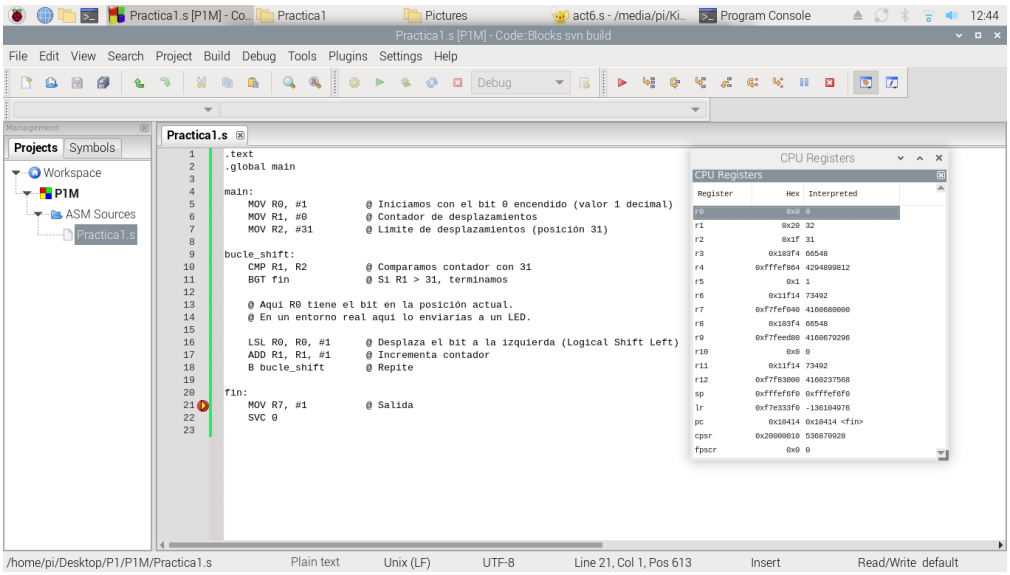


Figura 8: Verificación de registros en Code::Blocks para la Actividad 6, mostrando el estado final después de completar el bucle.

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 buques 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.

Actividad 8

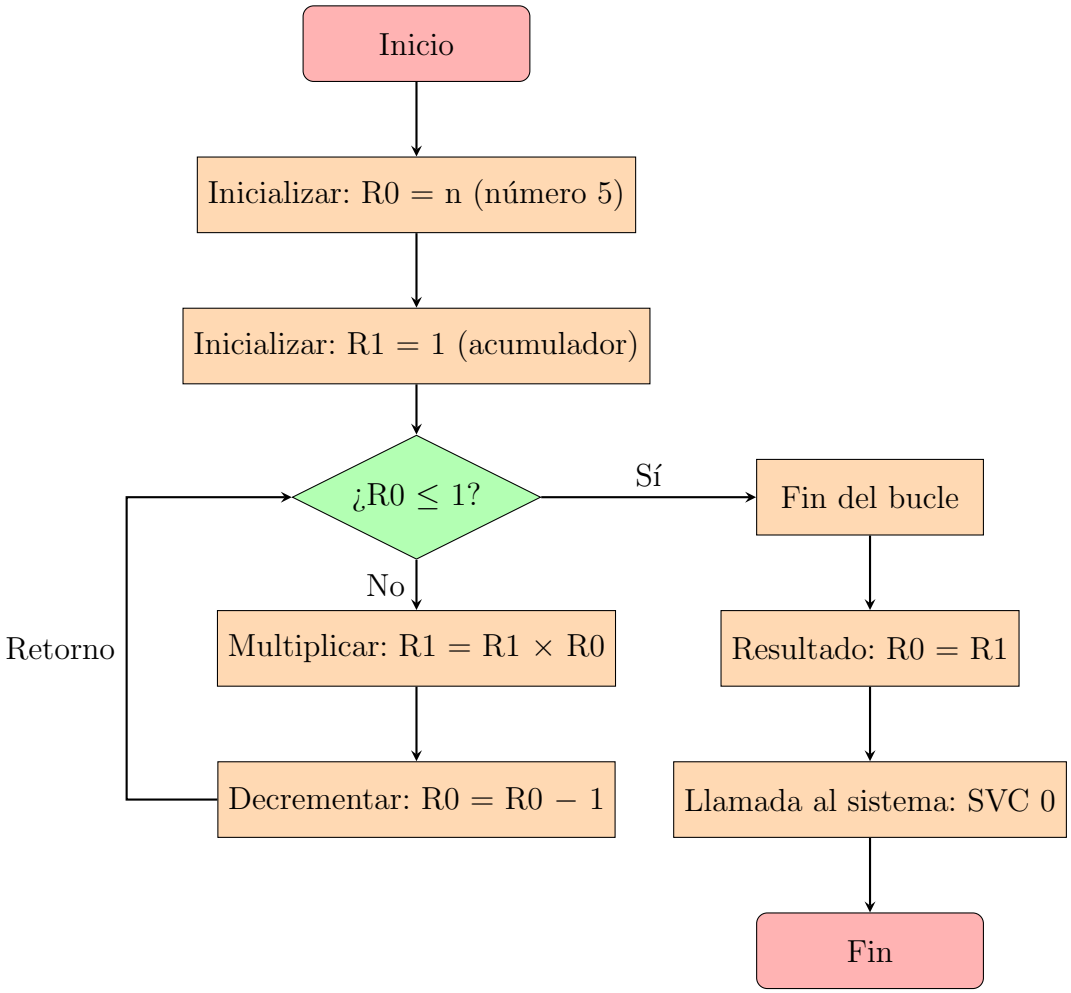
Escribir un programa que realice la suma de dos números de 64 bits y almacene el resultado en memoria empleando las direcciones que considere el resultado del acarreo en caso de existir.

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 $n! = n \times (n-1) \times (n-2) \times \cdots \times 1$. 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 7: Código de la Actividad 9

```
.text
.global main

main:
    MOV R0, #5          @ Número n al que calculamos factorial (ej. 5)
    MOV R1, #1          @ R1 guardará el resultado acumulado (inicia en 1)

loop_fact:
    CMP R0, #1          @ Compara n con 1
    BLE fin_fact        @ Si n <= 1, terminamos

    MUL R1, R1, R0      @ R1 = R1 * R0 (Acumulado * n)
    SUB R0, R0, #1      @ Decrementa n
    B loop_fact         @ Repite

fin_fact:
    MOV R0, R1          @ Mueve resultado final a R0
    MOV R7, #1          @ Salir
    SVC 0
```

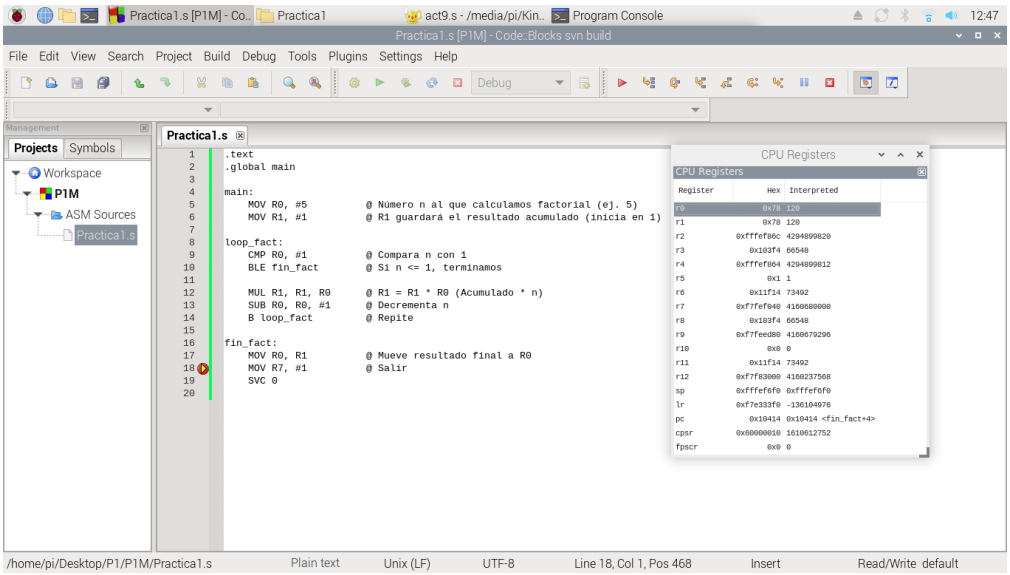


Figura 9: Verificación de registros en Code::Blocks para la Actividad 9, mostrando el resultado del factorial.

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:

2. Conclusiones:

- **Espinoza Matamoros Percival Ulises:**
- **Flores Colin Victor Jaziel:**
- **Lara Hernandez Angel Husiel:**