



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:

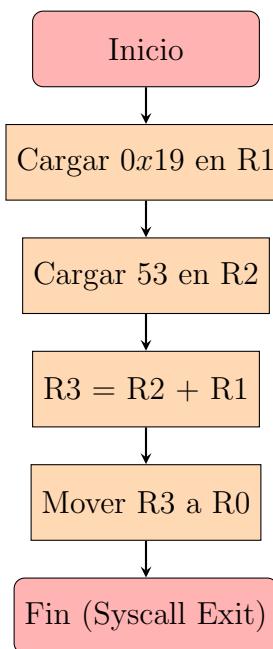
Aprender la estructura de los procesadores con arquitectura ARM, utilizar la plataforma Raspberry Pi, los entornos para programar; desarrollar algoritmos con las instrucciones en lenguaje ensamblador, controlar directamente los recursos del microprocesador; editar, compilar, ensamblar, simular y ejecutar 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 **direcciónamiento 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
2                         enlazador (linker)
3
4 _start:                  @ Punto de entrada del programa
5     MOV R1, #0x19        @ Carga el valor hexadecimal 19 (25 decimal)
6                         en el registro R1
7     MOV R2, #53          @ Carga el valor decimal 53 en el registro
8                         R2
9
10    ADD R3, R2, R1      @ Suma: R3 = R2 + R1. Resultado esperado: 78
11
12    MOV R0, R3          @ Mueve el resultado (78) al registro R0 (registro de retorno)
13
14    MOV R7, #1           @ Carga el valor 1 en R7. En Linux, 1 significa "Syscall Exit"
15
16    SVC 0               @ Supervisor Call: Llama al Kernel para
17                         ejecutar la salida

```

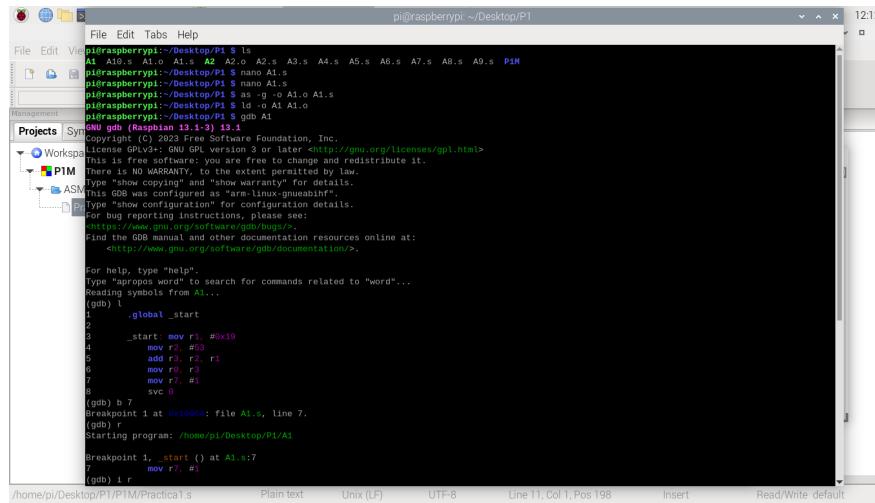


Figura 1: Proceso de ensamblado, enlazado e inicio de GDB en Raspberry Pi.

The screenshot shows the GDB interface running on a Raspberry Pi. The assembly code in the left pane is:

```
(gdb) l .global _start
1
2
3     _start: mov r1, #0x19
4     add r2, r2, r1
5     mov r0, r2
6     mov r7, #1
7     svc 0

(gdb) b 7
Breakpoint 1 at 0x10004: file A1.s, line 7.

(gdb) r
Starting program: /home/pi/Desktop/P1/A1

Breakpoint 1, _start () at A1.s:7
(gdb) i r
r0      0x00 0
r1      0x19 25
r2      0x35 53
r3      0x4e 78
r4      0x0 0
r5      0x0 0
r6      0x0 0
r7      0x0 0
r8      0x0 0
r9      0x0 0
r10     0x0 0
r11     0x0 0
r12     0x0 0
sp      0xffffef100 0xffffef160
bp      0x0 0
pc      0x10004 0x10004 <_start+16>
cpsr   0x10 16
fpscr  0x0 0
tpscr  <unavailable>

(gdb) c
Continuing.

(gdb)
```

The register dump shows R0=0, R1=25, R2=53, R3=78, and R7=0. The PC is at 0x10004, which corresponds to the start of the assembly code.

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

Análisis de resultados

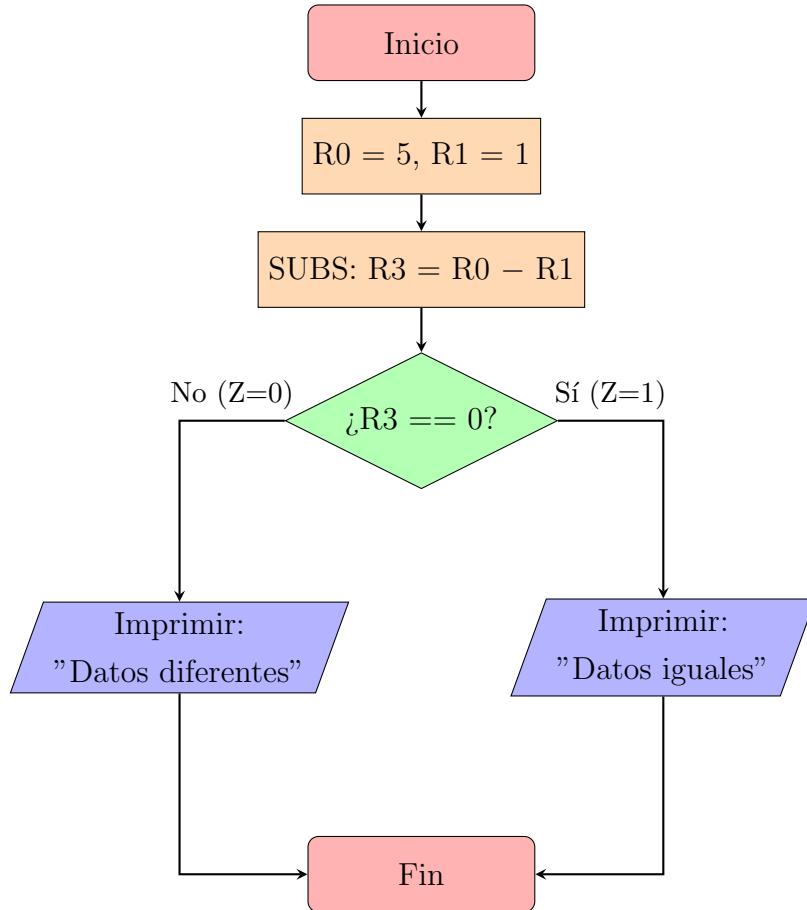
El programa utiliza el **modo de direccionamiento inmediato** para cargar constantes estáticas en los registros internos del procesador (**MOV R1, #0x19** y **MOV R2, #53**). El flujo interno de los datos envía el contenido de los registros R1 y R2 a la Unidad Aritmético Lógica (ALU) a través de la instrucción **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
8              actualiza banderas
9
10         BEQ igual          @ Si Z=1, salta a etiqueta 'igual'
11         BNE diferente       @ Si Z=0, salta a 'diferente'
12
13         igual:
14             MOV R0, #1          @ Descriptor de archivo 1 (Salida
15                 estandar / pantalla)
16             LDR R1, =texto1      @ Carga la direccion de 'texto1'
17             MOV R2, #14          @ Longitud del mensaje
18             MOV R7, #4           @ Syscall 4 (Write)
19             SVC 0
20             B fin              @ Salto al final
21
22         diferente:
23             MOV R0, #1          @ Descriptor de archivo 1
24             LDR R1, =texto2      @ Carga la direccion de 'texto2'
25             MOV R2, #17          @ Longitud del mensaje
26             MOV R7, #4           @ Syscall 4 (Write)
27             SVC 0
28
29         fin:
30             MOV R0, R3
31             MOV R7, #1           @ Syscall 1 (Exit)
32             SVC 0
```



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

The screenshot shows the QEMU debugger interface with the following details:

- File Edit Tabs Help** menu bar.
- Management** tab selected in the top-left corner.
- Projects** and **Syn** tabs in the top-left corner.
- Workspaces** section with a **PIM** workspace selected.
- ASM** section showing assembly code:

```
4     _start:
5         mov r0 #5
6         mov r1 #0x01
7
8         subs r3,r0,r1
9
10        beq igual
11
12        b 8
13
14 Breakpoint 1 at 0x1007c: file A2.s, line 8.
```
- P** section showing the program state:

```
(gdb) r
Starting program: /home/pi/Desktop/P1/A2

Breakpoint 1, _start () at A2.s:8
subs r3,r0,r1
(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          5
r1          0x1          1
r2          0x0          0
r3          0x0          0
r4          0x0          0
r5          0x0          0
r6          0x0          0
r7          0x0          0
r8          0x0          0
r9          0x0          0
r10         0x0          0
r11         0x0          0
r12         0x0          0
sp          0xffffef160  0xffffef160
lr          0x0          0
pc          0x1007c        0x1007c <_start+8>
cpsr        0x10          16
rpsr        0x0          0
tpidruru   <unavailable>
(gdb) [REDACTED]
```
- Registers** section showing register values.
- Memory** section showing memory dump.
- File** tab selected in the bottom-left corner.
- Plain text**, **Unix (LF)**, **UTF-8** buttons in the bottom center.
- Line 11, Col 1, Pos 198** status bar.
- Insert** button in the bottom right.
- ReadWrite default** status bar.

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.

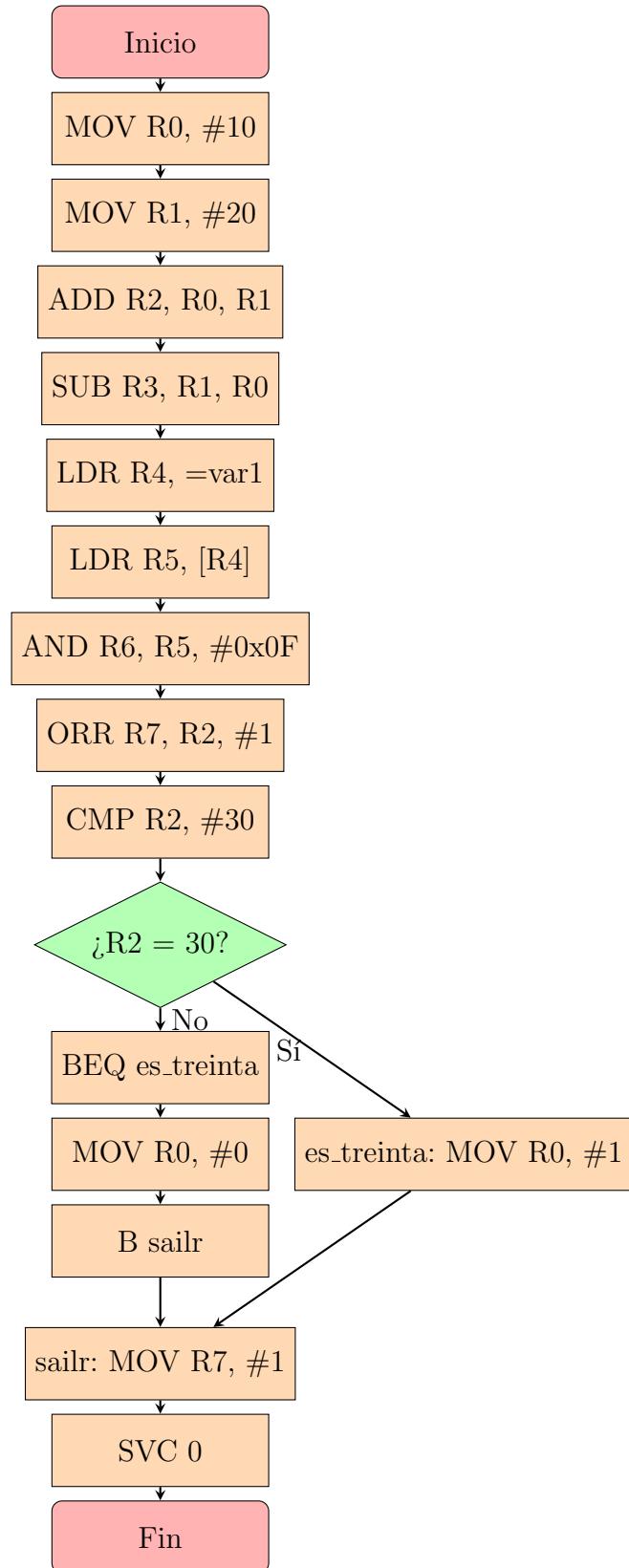
Actividad 3

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

- 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 OxAA          @ Definimos una variable con valor
3                         hexadecimal AA
4
5 .text
6 .global main
7
8 main:
9     MOV R0, #10           @ 1. MOV: Carga valor 10 en R0
10    MOV R1, #20            @ 2. MOV: Carga valor 20 en R1
11    ADD R2, R0, R1        @ 3. ADD: Suma 10 + 20 = 30 en R2
12    SUB R3, R1, R0        @ 4. SUB: Resta 20 - 10 = 10 en R3
13
14    LDR R4, =var1         @ 5. LDR (dirección): Carga dirección
15                         de var1 en R4
16    LDR R5, [R4]           @ 6. LDR (valor): Carga el valor OxAA de
17                         memoria en R5
18
19    AND R6, R5, #0x0F      @ 7. AND: Mscara para quedarse con la
20                         parte baja (0x0A)
21    ORR R7, R2, #1          @ 8. ORR: Operación lógica OR con 1
22                         (30 OR 1 = 31)
23
24    CMP R2, #30            @ 9. CMP: Compara si R2 es igual a 30
25    BEQ es_treinta         @ 10. BEQ: Salta si es igual (R2 = 30)
26
27    MOV R0, #0              @ Si no es igual, pone 0 en R0
28    B salir                @ 11. B: Salto incondicional (
29                         instrucción adicional)
30
31 es_treinta:
32     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

```

The screenshot shows the Code::Blocks IDE interface with the following details:

- Title Bar:** Practical1.s, pi@raspber, Practical1.s - /m..., Pictures, Pictures, Program C..., 12:21
- Menu Bar:** File, Edit, View, Search, Project, Build, Debug, Tools, Plugins, Settings, Help
- Toolbar:** Standard toolbar with icons for file operations.
- Project Explorer:** Shows a workspace named "P1" containing a project "PIM" which includes an "ASM Sources" folder with the file "Practical1.s".
- Code Editor:** Displays the assembly code for "Practical1.s" with comments explaining each instruction's purpose.
- Registers Window:** A floating window titled "CPU Registers" showing the state of various ARM registers. The register r9 is highlighted with a red box and has a tooltip "var1" above it. Other registers shown include r10, r11, r12, sp, lr, pc, and cpsr.
- Status Bar:** /home/pi/Desktop/P1/PIM/Practical1.s, Plain text, Unix (LF), UTF-8, Line 30, Col 1, Pos 871, Insert, Read/Write default

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.

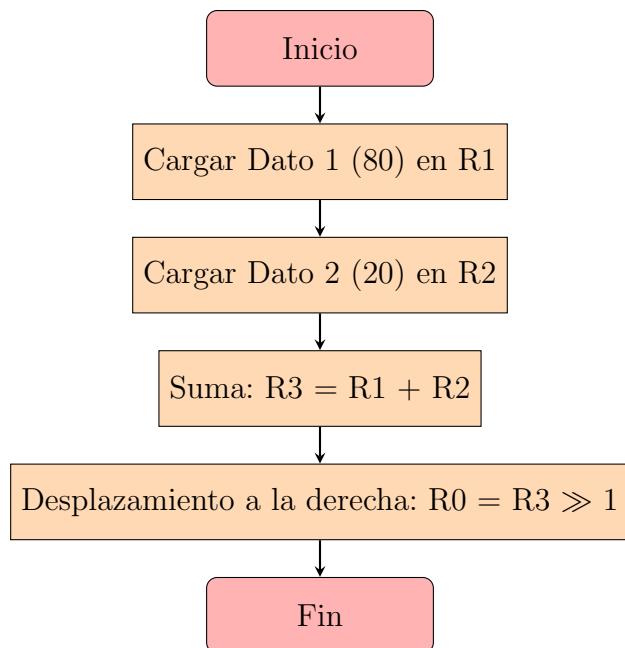
El depurador de Code::Blocks confirma la ejecución del programa y permite observar los valores finales en los registros, validando el correcto funcionamiento de la interacción con la ALU, el banco de registros, la memoria y el control de flujo condicional.

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

```

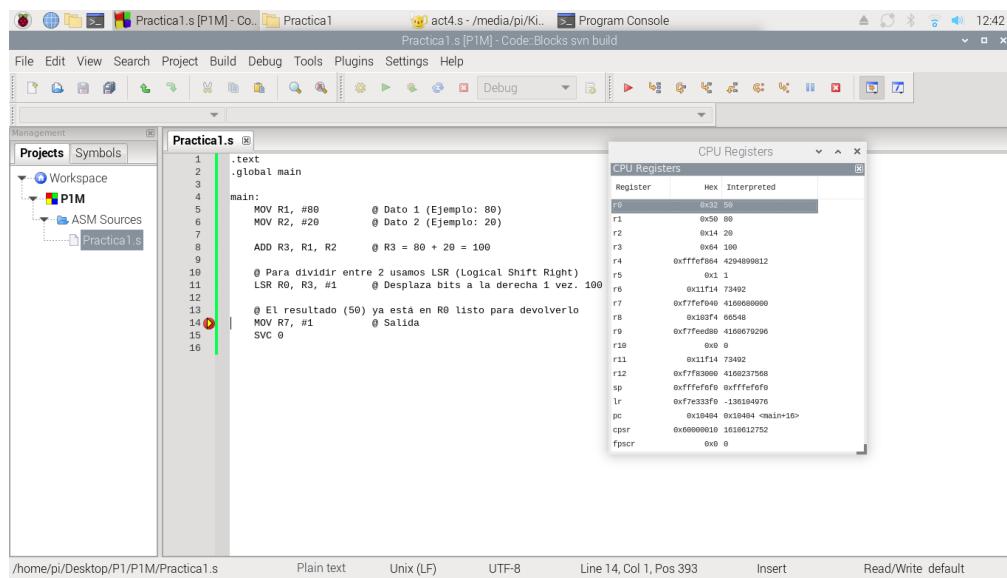


Figura 6: Verificación de registros en Code::Blocks para la Actividad 4, mostrando R0=0x32 (50).

Análisis de resultados

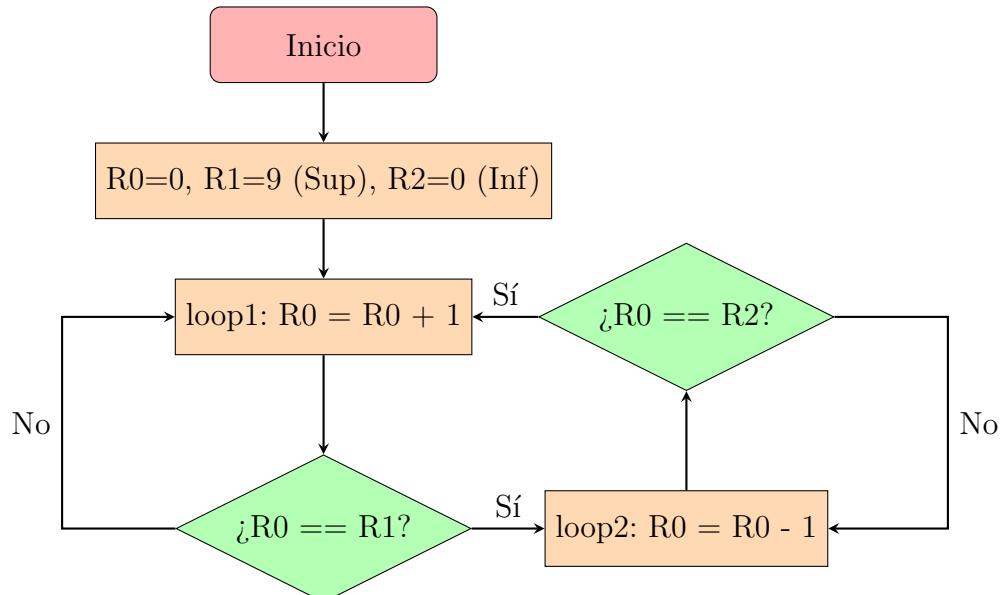
El objetivo se cumplió promediando dos números enteros de manera eficiente. En lugar de utilizar una instrucción de división pesada, se manipuló la arquitectura mediante un **Logical Shift Right (LSR)**. En la arquitectura ARM, el flujo de datos pasa a través de un desplazador de barril (*barrel shifter*) 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  /* Actividad 5: Bucle Ascendente y Descendente */
2
3  .text
4  .global main
  
```

```

5
6      main:
7          MOV R0, #0           @ R0 sera nuestro contador, inicia en 0
8          MOV R1, #9           @ R1 es el limite superior (9)
9          MOV R2, #0           @ R2 es el limite inferior (0)
10
11     loop1:                @ Etiqueta para subir
12         ADD R0, R0, #1       @ Incrementa R0 en 1
13         CMP R1, R0           @ Compara si llegamos al limite superior
14             (9)
15         BNE loop1           @ Si NO es igual, repite loop1
16
17     loop2:                @ Etiqueta para bajar
18         ADD R0, R0, #-1      @ Decrementa R0 en 1
19         CMP R2, R0           @ Compara si llegamos al limite inferior
20             (0)
21         BEQ loop1           @ Si es igual a 0, salta de nuevo a
22             loop1
23         B loop2              @ Si no es 0, sigue bajando

```

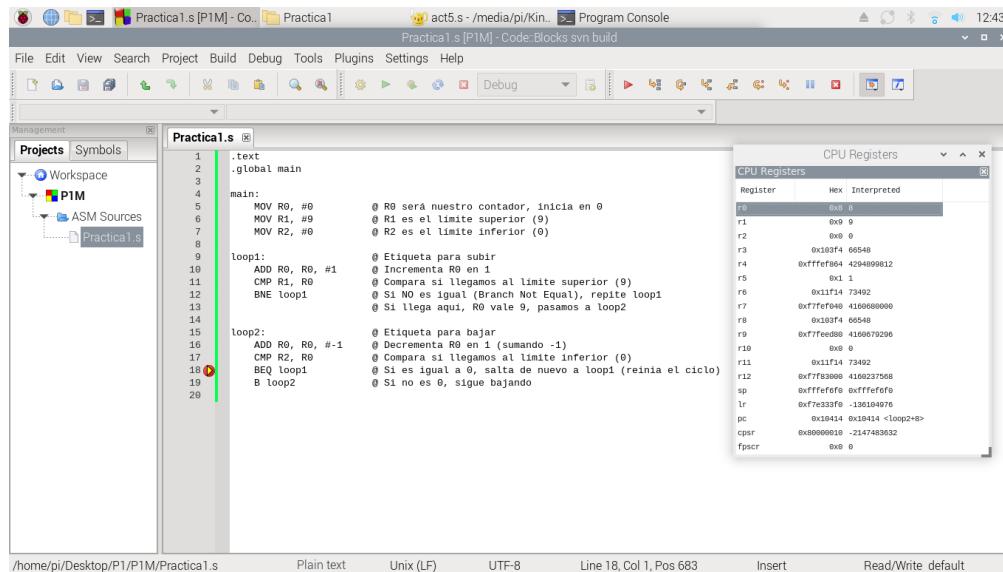


Figura 7: Monitoreo de bucle en Code::Blocks, mostrando a R0 en la iteración de bajada.



Análisis de resultados

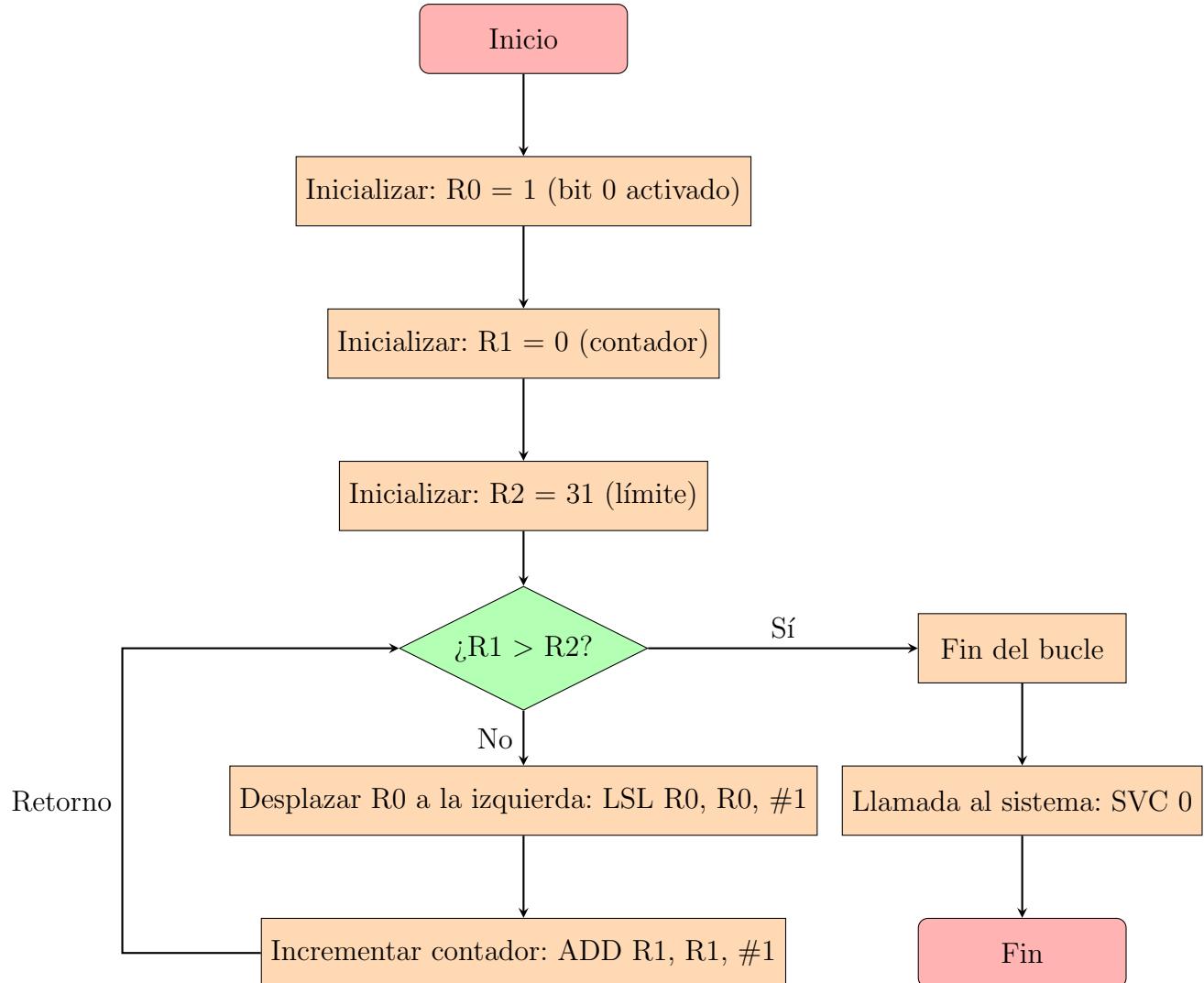
Este programa demuestra el funcionamiento interno del **Contador de Programa (PC)** ante instrucciones de bifurcación. El ciclo infinito es gobernado por la instrucción de comparación virtual **CMP**, que resta internamente los registros sin almacenar el resultado, solo para manipular el **CPSR**. Al 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 .global main
2
3 main:
4     MOV R0, #1          @ Inicializa R0 con el bit menos
5                   significativo activado (000...0001)
6     MOV R1, #0          @ Inicializa contador de posiciones en 0
7     MOV R2, #31         @ Establece el límite superior (bit más
8                   significativo en posición 31)

```

```

7
8 bucle_shift:
9     CMP R1, R2          @ Compara el contador con el límite
10    BGT fin             @ Si R1 > R2, salta a 'fin' (termina el
11      bucle)
12
13    @ Aquí R0 tiene el bit en la posición actual.
14    @ En un entorno real aquí lo enviarás a un LED.
15    LSL R0, R0, #1        @ Desplaza R0 un bit a la izquierda (el bit
16      activado avanza)
17    ADD R1, R1, #1        @ Incrementa el contador
18
19    B bucle_shift         @ Vuelve al inicio del bucle
20
21 fin:
22     MOV R7, #1           @ Prepara la llamada al sistema para salir (
23       sys_exit)
24     SVC 0                 @ Ejecuta la salida

```

The screenshot shows the Code::Blocks IDE interface. The main window displays the assembly code for 'Practical1.s'. The code includes comments explaining the logic: it initializes R0 to 1, sets R1 to 31, and R2 to 31. It then enters a loop where it compares R1 with R2, increments R1, and shifts R0 left by one bit. Once R1 exceeds R2, it exits the loop and prepares to exit the system via a SVC 0 instruction. The right side of the interface features a 'CPU Registers' window, which lists all general-purpose registers (r0 to r15) along with their hex and interpreted values. The interpreted values show the state of the registers after the program has run, reflecting the logic described in the assembly code.

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.



Actividad 8

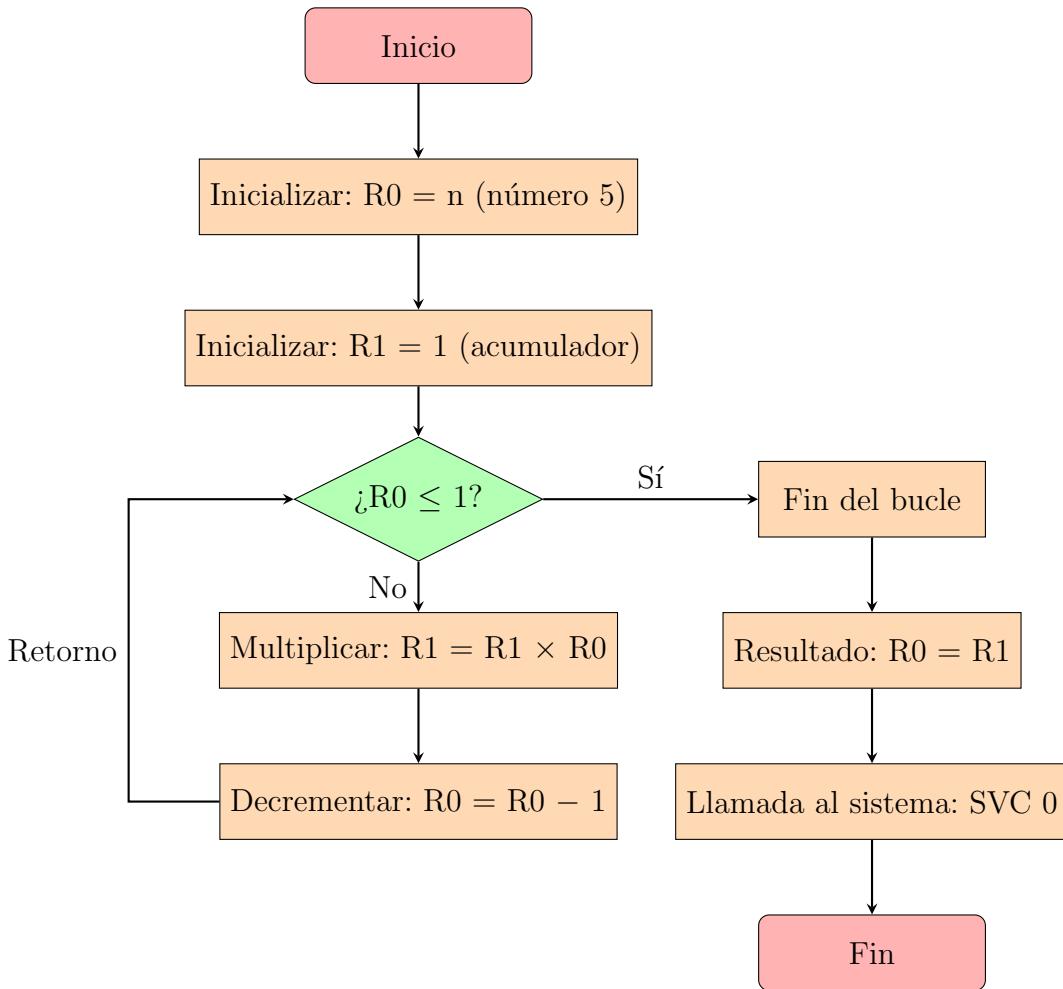
Escribir un programa que realice la suma de dos números de 64 bits y almacene el resultado en memoria empleando las direcciones que considere el resultado del acarreo en caso de existir.

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 $n! = n \times (n - 1) \times (n - 2) \times \dots \times 1$. 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 7: 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 guardar el resultado acumulado (
8                 inicia 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
```

The screenshot shows the Code::Blocks IDE interface. On the left, the project tree shows 'PIM' and 'Práctica1.s'. The main window displays the assembly code for 'Práctica1.s' with comments explaining the logic of the factorial calculation. The right side shows the 'CPU Registers' window with the following table:

Register	Hex	Interpreted
r0	0x78	128
r1	0x78	128
r2	0xffffef80	4294899980
r3	0x103f4	65548
r4	0xffffef804	42948999812
r5	0x1	1
r6	0x11f14	73402
r7	0x7fe1640	4160686000
r8	0x103f4	65548
r9	0xfffffe00	4160679296
r10	0x0	
r11	0x11f14	73402
r12	0x7fe00000	4160237568
sp	0xffffef00	0xffffef00
lr	0x7e333f9	-130164976
pc	0x10414	0x10414 <fin_fact+4>
cpsr	0x00000010	1610612752
fpscr	0x0	

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:



Referencias

- Anaya, R. (s.f.). *Manual de recursos y aplicaciones Plataforma Raspberry Pi*. <https://odin.filb.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.^a 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>