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:

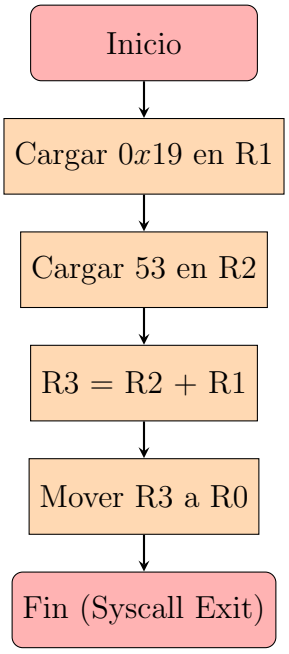
Aprender la estructura de los procesadores con arquitectura ARM, utilizar la plataforma Raspberry Pi, los entornos para programar; desarrollar algoritmos con las instrucciones en lenguaje ensamblador, controlar directamente los recursos del microprocesador; editar, compilar, ensamblar, simular y ejecutar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

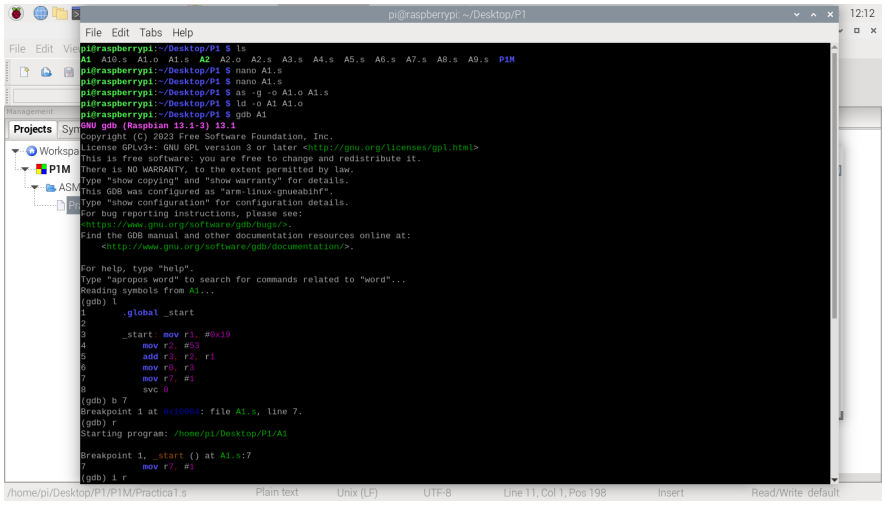


Figura 1: Proceso de ensamblado, enlazado e inicio de GDB en Raspberry Pi.

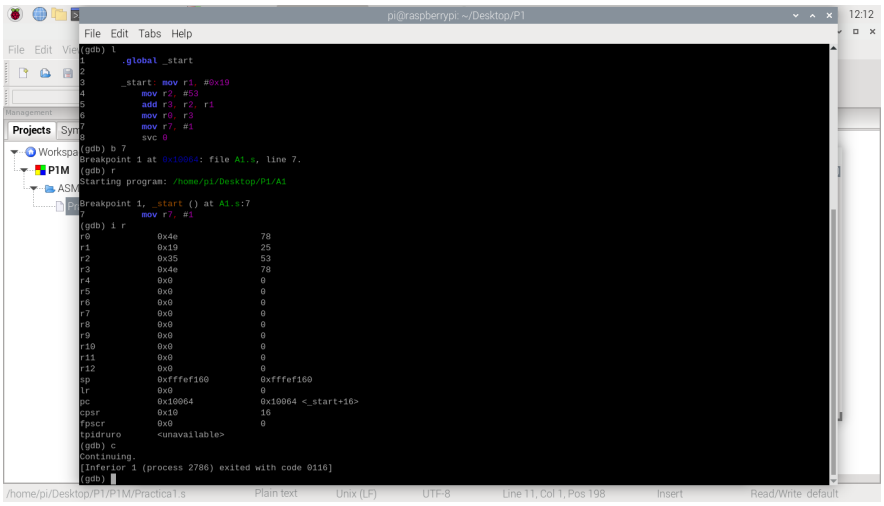


Figura 2: Inspección de registros en GDB mostrando R0 y R3 con el valor de 78 (0x4E).

Análisis de resultados

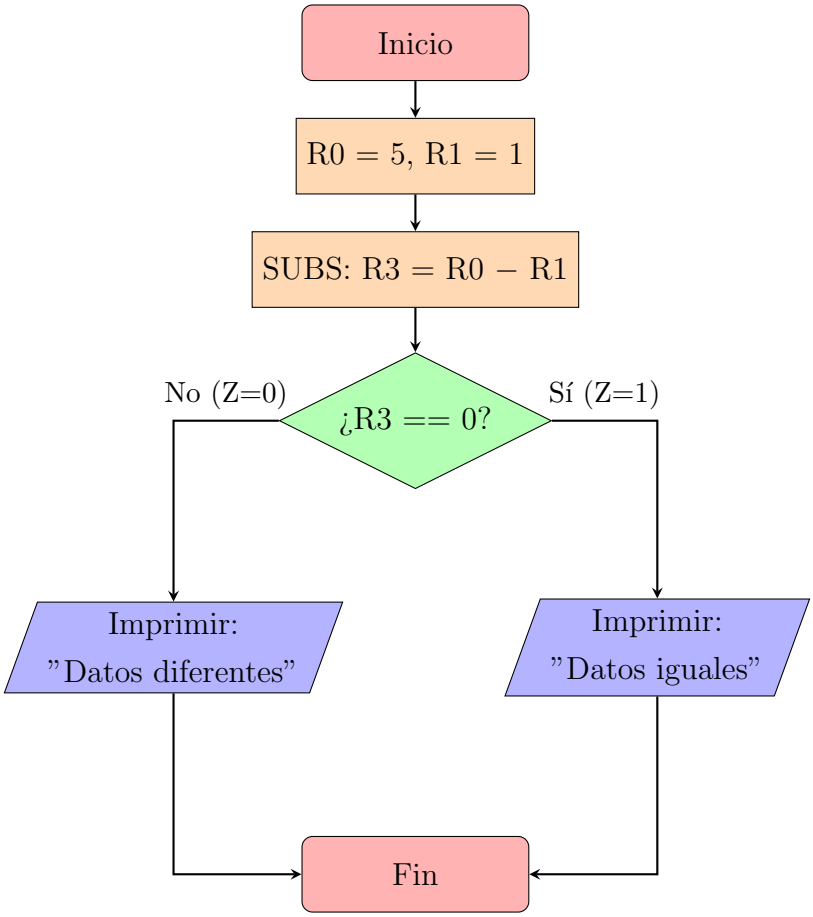
El programa utiliza el **modo de direccionamiento inmediato** para cargar constantes estáticas en los registros internos del procesador (`MOV R1, #0x19` y `MOV R2, #53`). El flujo interno de los datos envía el contenido de los registros R1 y R2 a la Unidad Aritmético Lógica (ALU) a través de la instrucción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

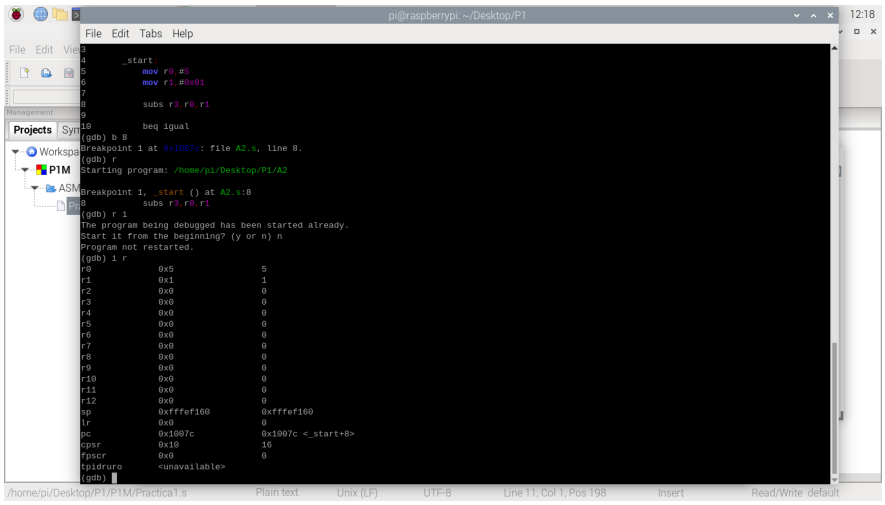


Figura 3: Inspección de registros antes del salto condicional.

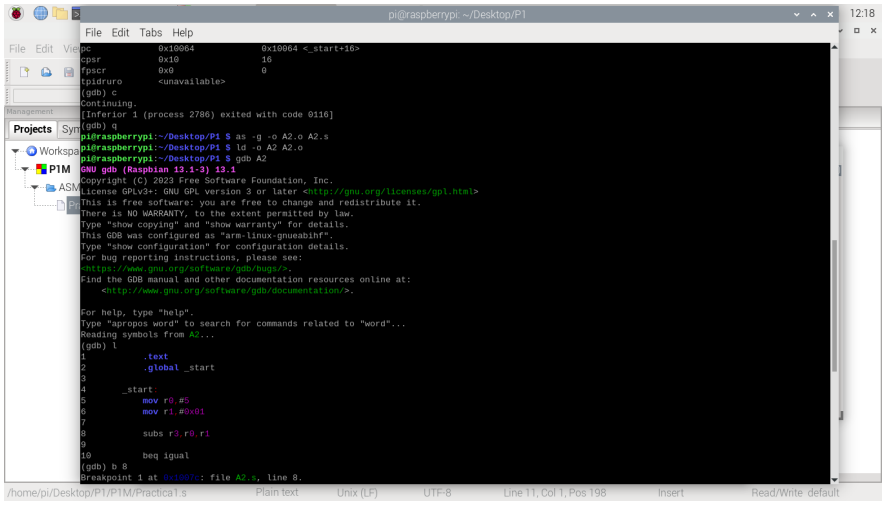


Figura 4: Ejecución del programa evaluando la condición de desigualdad.

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::Block, seleccionar 10 instrucciones, formalizar un programa; comprobar el funcionamiento (agregar las directivas correspondientes).

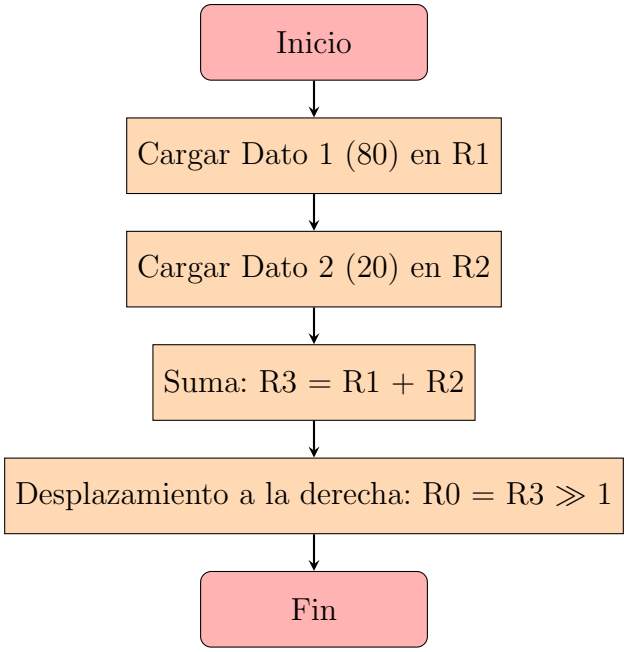
a. Reportar el resultado esperado y el obtenido.

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 3: 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
```

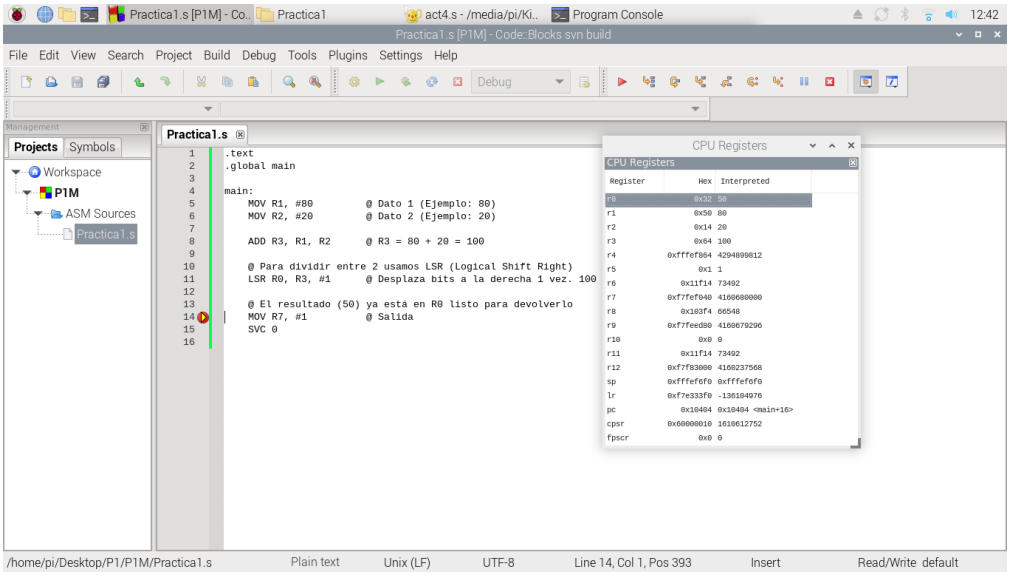


Figura 5: Verificación de registros en Code::Blocks para la Actividad 4, mostrando R0=0x32 (50).

Análisis de resultados

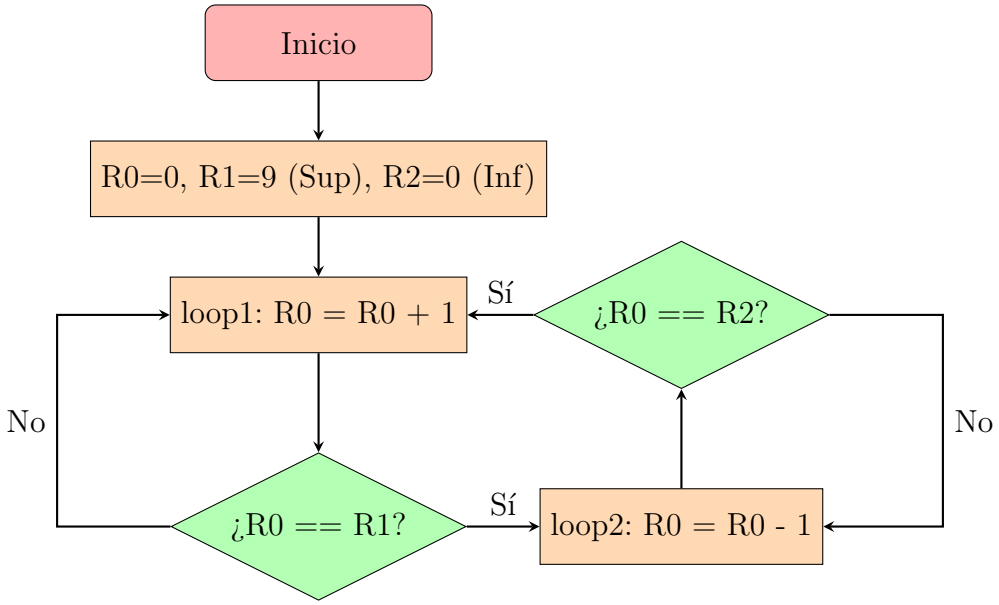
El objetivo se cumplió promediando dos números enteros de manera eficiente. En lugar de utilizar una instrucción de división pesada, se manipuló la arquitectura mediante un **Logical Shift Right (LSR)**. En la arquitectura ARM, el flujo de datos pasa a través de un desplazador de barril (*barrel shifter*) 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 4: 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
            (9)
14      BNE loop1           @ Si NO es igual, repite loop1
15
16      loop2:                  @ Etiqueta para bajar
17      ADD R0, R0, #-1     @ Decrementa R0 en 1
18      CMP R2, R0          @ Compara si llegamos al limite inferior
            (0)
19      BEQ loop1           @ Si es igual a 0, salta de nuevo a
           loop1
20      B loop2             @ Si no es 0, sigue bajando
```

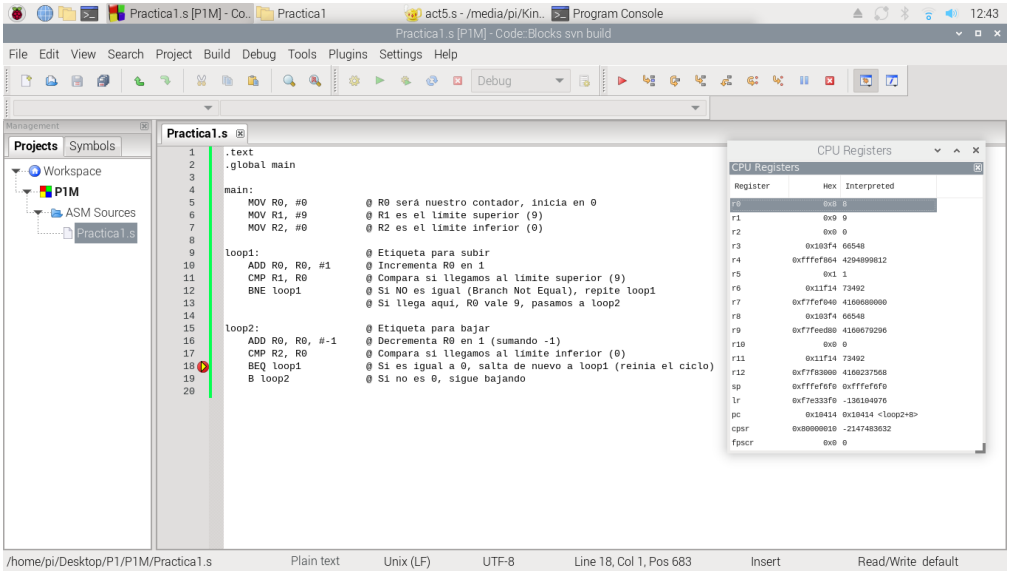


Figura 6: Monitoreo de bucle en Code::Blocks, mostrando a R0 en la iteración de bajada.

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.

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.

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 Rasberry Pi*. https://odin.fi-b.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