Lab 4: Introduction to ARM assembly language with Loop, Function

Goals

1. Understanding the fundamental concepts of Loop & Function
2. Able to write simple assembly programs using ARMv7
3. Develop familiarity with CPULator

Theory:

Loop:

In ARM assembly, loops can be imitated using **conditional branches** and **compare instructions**. The basic idea is to perform a series of operations repeatedly until a certain condition is met, much like how loops work in high-level languages such as `for`, `while`, or `do-while`.

Let's explore how to imitate different types of loops in ARM assembly:

For Loop

A **for loop** typically has:

- Initialization
- Condition check
- Update/Increment
- Loop body

Example: For Loop in C

```
for (int i = 0; i < 10; i++) {
        // Loop body
        }
```

Imitation in ARM Assembly:

```
MOV R0, #0          ; Initialize the counter i = 0
MOV R1, #10         ; Set the limit (loop until i < 10)

loop_start:

CMP R0, R1          ; Compare i with 10
BGE loop_end        ; If i >= 10, exit the loop

      ; Loop body (whatever you want to repeat goes here)

 ADD R0, R0, #1         ; Increment i
 B loop_start           ; Repeat the loop

loop_end:
      ; Loop ends here
```

Explanation:

- **MOV R0, #0**: Initialize the counter (i = 0).
- **MOV R1, #10**: Set the loop limit (i < 10).
- **CMP R0, R1**: Compare i with 10.
- **BGE loop_end**: If i >= 10, branch to the label loop_end to exit the loop.
- **ADD R0, R0, #1**: Increment i.
- **B loop_start**: Branch back to loop_start to repeat the loop.

Function:

To use the concept of function in arm assembly, we basically use the branch. Specifically, we use BL (Branch with link) opcode when using function.

BL (Branch with Link): It branches like an unconditional branch but stores the address of the next instruction in the link register. So, after the function task, we can return back to the caller.

Creating a function to add two numbers:

The code will look like this:

```
.text
_start:

MOV r0, #1        @first number
MOV r1, #3        @second number
BL add_num        @ calling the function



add_num:          @function name as Label
ADD r2, r0, r1          @ doing the sum
BX LR             @ sending back to the next line of BL
```

Points to be noted:

- Functions will be implemented using branch
- You must use BL and BX (Branch Exchange) to return back to the caller
- Be careful about overwriting values. To avoid, you need to use stack segment. (Slides)

Stack Segment:

The stack is a memory region within the program/process. This part of the memory gets allocated when a process is created. We use Stack for storing temporary data, such as local variables of some functions and environment variables, which help us to transition between the functions, etc.

To avoid the value overwriting while we use the function, we can use Stack Segment.

Stack's 2 operations:

1. PUSH

   PUSH instruction is used to store the registers in the stack segment. You need to follow the syntax given below:

   ```
   PUSH {registers}
   ```

   Example:

   ```
   PUSH {R0, R1}
   ```

2. POP

   POP instruction is used to remove the register's values in the stack segment and move them to the register. You need to follow the syntax given below:

   ```
   POP {registers}
   ```

   Example:

   ```
   POP {R0, R1}
   ```

Declaring a constant in assembly code:

The declaration must follow the given syntax:

```
.equ constant_name, value
.equ constant_name, address_value (must be in HEX)
```

Example:

Suppose, you want to declare a constant with 7 value named Y. The code will be:

```
.equ Y, 7  @works like define directives in C
```

Usage:

1. Used to store a value that will never be changed
2. Used to store address when we use display or LEDs in **CPULATOR**

CPULator Peripherals:

In **CPUlator**, peripherals are simulated hardware components that allow you to interact with the ARM system by performing input/output operations or interacting with memory-mapped hardware devices. The ARM version of CPUlator typically includes peripherals found in embedded systems such as LEDs, switches, 7-segment displays, timers, and UARTs.

Common Peripherals in CPUlator for ARM

1. **7-Segment Display**
   - **Description**: A simulated 7-segment display where each segment can be turned on or off to display numbers or letters.
   - **Memory Address**: The 7-segment display is memory-mapped, so you can write to specific memory addresses to control the display.
   - **Usage**:
     - You can write specific hex values to the memory-mapped address to display numbers.
     - Typically used to display single-digit values (0–9).

Assembly program example:

```
LDR R0, =0xFF200020   @ Address of the 7-segment display
MOV R1, #0x3F         @ Value to display "0" on the 7-segment
STR R1, [R0]          @ Write the value to the 7-segment display
```

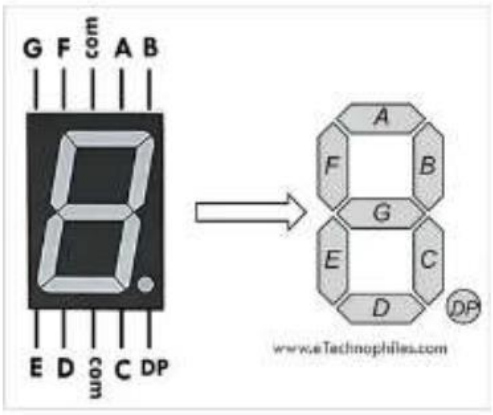
Same Code But more efficient:

```
.equ SEGMENT_DISPLAY_ADDR, 0xFF200020   @ Address of the 7-segment display
.equ DISPLAY_0_VALUE, 0x3F              @ Value to display "0" on the 7-segment

_start:
   LDR R0, =SEGMENT_DISPLAY_ADDR  @ Load the address of the 7-segment display into R0
   MOV R1, #DISPLAY_0_VALUE       @ Load the value to display "0" into R1
   STR R1, [R0]                   @ Write the value to the 7-segment display

.end
```

7-Segment Display Recap:



In ARM assembly, the value 0x3F corresponds to the hexadecimal representation of the binary pattern that lights up specific segments of a 7-segment display to form the number "0." Let's break this down:

How a 7-Segment Display Works:

A 7-segment display consists of seven individual segments (labeled A to G and a dot denoting by H) that can be turned on or off to display numbers and some letters. Each segment corresponds to a bit in a binary value, where:

- A bit of 1 means the segment is lit up (on).
- A bit of 0 means the segment is off.

Truth Table for 7-Segment Display (Hex Values for Digits 0-9):

Digit	Segments (A-G H is always Zero)	Binary	Hex
0	A, B, C, D, E, F	0011 1111	0x3F
1	B, C	0000 0110	0x06
2	A, B, D, E, G	0101 1011	0x5B
3	A, B, C, D, G	0100 1111	0x4F
4	B, C, F, G	0110 0110	0x66
5	A, C, D, F, G	0110 1101	0x6D
6	A, C, D, E, F, G	0111 1101	0x7D
7	A, B, C	0000 0111	0x07
8	A, B, C, D, E, F, G	0111 1111	0x7F
9	A, B, C, D, F, G	0110 1111	0x6F

LEDs

- **Description**: CPUlator simulates a bank of general-purpose LEDs. You can control these LEDs by writing to specific memory addresses.
- **Memory Address**: The LEDs are also memory-mapped.
- **Usage**:
  - Each bit in the control register corresponds to a different LED.
  - Setting a bit to 1 turns the corresponding LED on, and 0 turns it off.

Assembly program example:

```
LDR R0, =0xFF200000    @ Address of the LED control
MOV R1, #0xFF          @ Turn on all LEDs
STR R1, [R0]           @ Write the value to the LED register
```

Here is an ARM assembly program that cycles through the numbers 1 to 15 and displays each number in binary on the LEDs. The LEDs are controlled by writing a binary value to the memory-mapped LED register. Assuming the memory address for the LEDs is 0xFF200000, the program will write the binary values of numbers 1 through 15 to this address in a loop, simulating a binary counter on the LEDs.

Assembly Code:

```
  .equ LED_ADDR, 0xFF200000   @ Address for the LEDs

  .global _start              @ Define the starting point of the program

_start:
  MOV R1, #1                  @ Initialize R1 with 1 (starting number)

loop:
  LDR R0, =LED_ADDR           @ Load the address of the LEDs into R0
  STR R1, [R0]                @ Store the value of R1 (in binary) to the LED address

  ADD R1, R1, #1              @ Increment R1 by 1
  CMP R1, #16                 @ Compare R1 with 16 (to loop through 1 to 15)
  BEQ reset_counter           @ If R1 == 16, reset the counter

  B loop                      @ Branch back to loop

reset_counter:
  MOV R1, #1                  @ Reset the counter back to 1
  B loop                      @ Start the loop again

done:
  B done                      @ Loop forever (for safety)
```

Explanation:

1. **Constants**:
    - The **.equ** directive is used to define the memory address for the LEDs (LED_ADDR), which is set to 0xFF200000.
2. **Initialization**:
    - The program starts by initializing register R1 to 1, which is the first number to be displayed.
3. **Loop**:
    - The main loop loads the LED memory address into R0 and then stores the value of R1 (representing the current number) at the LED address.
    - After displaying the current number, R1 is incremented by 1, and the program checks if the value in R1 has reached 16.
    - When R1 reaches 16, the program resets it back to 1, creating a continuous loop of numbers from 1 to 15.
4. **LED Control**:
    - Each time R1 is written to the LED memory address, the value is shown in binary on the LEDs. For example, if R1 = 5, the LEDs will show 0000000000000101 in binary (assuming a 16-LED display).

Memory Address and Peripherals:

- **LED Memory Address**: 0xFF200000 is the base address of the memory-mapped LEDs. Writing a binary value to this address turns the corresponding LEDs on or off.

How It Works:

- The program repeatedly increments the number from 1 to 15, displaying each in binary on the LEDs.
- After reaching 15, the program resets the counter to 1 and repeats the process.

This program simulates a binary counter on the LEDs using ARM assembly. The numbers cycle from 1 to 15, and each is displayed as a binary value on the LEDs.

Lab Demo

Lab 5: Introduction to Microprocessor Interfacing

Name:
ID:
Section:

1. How do I find the addresses of all the LEDs, switches, push buttons, and Seven segments? Write ARM Assembly Code to drive them all
2. Write a Code for ARM assembly that Calculates the Factorial of a number and shows that number's Binary Value using LEDs