Programming Assignment 1

Based on the topics covered in Module 1, these assignments are designed to test your understanding of fundamental ARM assembly concepts, including arithmetic operations, control flow, memory access, and bit manipulation.

Submission Guidelines:

To complete this assignment set, please follow these steps:

1. File Organization:
    - Write the code for each problem in a separate ARM assembly file.
    - Name your files sequentially: `problem1.s`, `problem2.s`, `problem3.s`, and so on.
2. Video Demonstration:
    - Record a short video (maximum of 5 minutes) where you give a brief run-through of your solution for each problem and show the final result in the simulator.
    - Upload this video to YouTube as an "Unlisted" video.
3. Repository & Documentation:
    - Create a public repository on a platform like GitHub to host your work.
    - Upload all your `.s` files.
    - In the repository, create a `README.md` file. This file must contain:
        - A short description of your insights or any challenges you faced while working on the assignments.
        - The link to your unlisted YouTube video.
4. Final Submission:
    - Submit the URL to your public repository.

Problem 1: Factorial Calculation

Objective: Write an ARM assembly program to calculate the factorial of a given non-negative integer.

Problem Statement:
The factorial of a non-negative integer n, denoted by n!, is the product of all positive integers less than or equal to n. For example, 5! = 5 * 4 * 3 * 2 * 1 = 120. By definition, 0! = 1.
Your task is to write a program that takes an integer n from a register (e.g., r0) and computes the result, storing it in another register (e.g., r1).

Input:

- r0: A non-negative integer n. You can assume n will be small enough that the result fits within a 32-bit register (i.e., n <= 12).

Output:

- r1: The computed value of n!.

Hint:
This problem can be solved iteratively (using a loop) or recursively (by having a function call itself). If you choose the recursive approach, remember to use the stack (PUSH, POP) to save the return address (lr) and any registers your function modifies.

Problem 2: Euclidean Algorithm for GCD

Objective: Implement the Euclidean algorithm in ARM assembly to find the greatest common divisor (GCD) of two positive integers.

Problem Statement:
The greatest common divisor (GCD) of two integers is the largest positive integer that divides both numbers without leaving a remainder. The Euclidean algorithm is an efficient method for computing the GCD. A modern version of the algorithm is as follows:

While **b ≠ 0**:
  Set **temp = b**
  Set **b = a mod b**
  Set **a = temp**

The GCD is the final value of **a**.

Input:

- r0: The first positive integer, a.
- r1: The second positive integer, b.

Output:

- r0: The GCD of a and b.

Hint:
Use the `MOD` operation carefully—since ARM doesn't have a direct modulo instruction, you'll need to use UDIV (see documentation) instruction combined with multiply and subtract operations to implement modulo.

Problem 3: Bitwise Parity Checker

Objective: Write an ARM assembly program to check the parity of a 32-bit number. Parity is the property of whether the number of set bits (bits with a value of 1) is even or odd.

Problem Statement:
Given a 32-bit integer, determine if it contains an even or odd number of 1s.
For example:

- The number 5 (binary 0101) has two 1s, so its parity is **even**.
- The number 7 (binary 0111) has three 1s, so its parity is **odd**.

Input:

- r0: A 32-bit integer.

Output:

- r1: Set to 0 if the number of set bits is **even**.
- r1: Set to 1 if the number of set bits is **odd**.

Hint:
To count bits, you can repeatedly test the least significant bit (using `AND` with `#1`) and then shift right (`LSR`) until the number becomes zero. Use a loop counter in another register to track parity. At the end, store parity as `0` (even) or `1` (odd) in `r1`.

Problem 4: Swap Nibbles in a Byte

Objective: Write a program that takes an 8-bit value (a byte) and swaps its upper 4 bits (most significant nibble) with its lower 4 bits (least significant nibble).

Problem Statement:
A byte can be seen as two 4-bit parts called nibbles. Your task is to swap these two parts.
For example:

- If the input is 0xA5 (binary 1010 0101), the output should be 0x5A (binary 0101 1010).

Input:

- r0: A register containing the 8-bit value. You can assume the upper 24 bits are zero.

Output:

- r1: The result with the nibbles swapped.

Hint:
Use bit masking and shifts. Mask the upper nibble with `AND rX, r0, #0xF0` *and shift it right (*`LSR #4`*). Mask the lower nibble with* `AND rY, r0, #0x0F` *and shift it left (*`LSL #4`*). Then combine the results using* `ORR`*. Remember to store the final value in* `r1`.

Problem 5: Find the Maximum Value in an Array

Objective: Write an ARM assembly program to find the largest integer in a given array of numbers.

Problem Statement:
An array of signed 32-bit integers is defined in the .data section of your program. You need to load the address of this array, iterate through its elements, and find the maximum value.
Input:

- r0: The starting address of the array.
- r1: The number of elements in the array.

Output:

- r2: The largest integer found in the array.

Setup:
You will need to define an array in your code like this:

```
.data
my_array: .word 4, 5, 9, 1, 0, -2, 3
array_size: .word 7
```

Hint:
Load the base address of the array in `r0` and loop through elements using `LDR` with post-increment addressing. Initialize `r2` with the first element, then compare each subsequent element with `CMP` and update `r2` if a larger value is found using `MOVGT`. Stop when you've processed the count in `r1`.