CG2028 Assignment 1

Su Menghang, Vincent (A0272102X)
Chan Xu Ming Ethan (A0273774R)

Introduction

This assignment implements an Infinite Impulse Response (IIR) filter using ARMv7-M assembly language for the STM32L4S5 microcontroller. The task involves writing an assembly function `int iir(int N, int* b, int* a, int x_n)` that processes digital signals according to the IIR filter equation:

$$y[n] = \frac{1}{a_0}\left(\sum_{i=0}^{N} b_i \cdot x[n-i] - \sum_{i=1}^{N} a_i \cdot y[n-i]\right) \tag{1}$$

The implementation must handle filter coefficients `b[]` and `a[]`, maintain internal state for delayed input `x[n-i]` and output `y[n-i]` values, and return the current filter output. The assembly function is called from a C program and must work for any filter order N up to N_MAX = 10.

Key challenges include efficient memory management for storing previous values, proper parameter passing through ARM registers (R0-R3), and optimizing the implementation for better performance than the reference C code provided.

Question 1

Knowing the starting address of a 2-d array `Arr[][]`, the memory address of element `Arr[A][B]` with index A and B starting from 0 is:

$$\text{Address} = \text{Base} + (A \times N + B) \times 4 \tag{2}$$

where each integer takes 4 bytes and N is the number of columns.

Question 2

After executing `BX LR`, the Link Register points to the instruction immediately after the call to `foo` in `main.c`, i.e. the `printf` at Line 36. This is because `BL foo` stored the return address in LR before branching. Although LR was modified inside `foo` by another `BL`, the `PUSH {LR}` and `POP {LR}` preserved the original return address.

Question 3

(i) Without `PUSH {R14}` and `POP {R14}`, the original LR (pointing to the `printf` in Line 36) gets overwritten by the inner `BL SUBROUTINE`. At the end, `BX LR` branches back into the instruction itself, causing an infinite loop / incorrect return.

(ii) With `PUSH {R14}` and `POP {R14}`, the original LR is saved on the stack and restored after the subroutine call. Thus `BX LR` correctly returns execution to the `printf` in Line 36, and the program behaves correctly.

Question 4

If the number of values exceeds the available registers, excess or less frequently used variables can be stored in memory using the `STR` instruction. When needed, they can be reloaded into registers using the `LDR` instruction.

Question 5

Machine code representations:

No.	Instruction	Binary	Hex
1	`ADD R12, R12, R6`	0b00000000100011001100000000000110	0x008CC006
2	`LDR R4, [R1]`	0b00000101000100010100000000000000	0x05114000
3	`BLT EXIT`	0b10111000000000000000000000001100	0xB800000C
4	`MUL R6, R6, R8`	0b00000000000000000110100000000110	0x00006806
5	`STR R4, [R5]`	0b00000101000001010100000000000000	0x05054000

Question 6

A modified datapath design includes:

- Adding a hardware multiplier block `MUL` with inputs `Mult_In_A`, `Mult_In_B`, and output `Mult_Out_Product`.
- Adding a third read port (A3/RD3) in the register file to supply the additional source operand for MLA.
- For MUL: Product routed directly into the Result MUX and written to Rd.
- For MLA: Product routed into an adder with Ra (from RD1) before Result MUX, then written to Rd.
- Control logic updated to distinguish between `MUL` and `MLA`.

Program Logic

- *In SUBROUTINE:*
  - `R4--R12` are pushed onto the stack.

  - `b[0]` and `a[0]` are loaded; result of `x_n * b[0] / a[0]` stored in `R12`.
  - Loop counter $k \leftarrow N - 1$; index i (current position in circular buffer) and j (coefficient index) initialized.
  - Circular buffer index loaded from memory to track current position.

- *In LOOP:*
  - Compare k with 0; exit if less than zero.
  - Calculate previous indices using modular arithmetic: `prev_idx = (current_idx - j - 1) mod N`.
  - Load `x_array[prev_idx]`, `y_array[prev_idx]`, `b[j+1]`, `a[j+1]`.
  - Compute `x_b = b[j+1] * x_array[prev_idx]`, `y_a = a[j+1] * y_array[prev_idx]`.
  - Calculate `(x_b - y_a) / a[0]` and add to running sum in `R12`.
  - Update counters: decrement k, increment j.

- *In EXIT:*
  - Move result from `R12` to `R0`.
  - Scale by dividing by 100.
  - Store new `x_n` and `y_n` into arrays at current index i.
  - Increment i with wrap-around and store back to memory.
  - Restore registers with `POP`.
  - Return with `BX LR`.

Improvements Made

- Original C code shifts arrays `x_store` and `y_store`, requiring $O(N)$ updates per iteration.
- Optimized assembly avoids shifting by using a circular buffer with index counter `i`.
- New values stored directly at `x_store[i]` and `y_store[i]` without moving other elements.
- Memory operations reduced from $2N$ to 2 per iteration (67% reduction for N=4).
- Instructions reordered to minimize pipeline stalls and improve execution efficiency.
- Register allocation strategy minimizes memory access by keeping frequently used values in registers.
- Loop structure optimized to reduce branch overhead and improve cache performance.

Appendix: Contributions

Name	Contribution
Vincent	Assembly code and report (focus on code)
Ethan	Assembly code and report (focus on report)