NUS ECE CG2028 Computer Organization
Assignment 1 – Semester 1, AY 2024/25
ARM v7-M Assembly Language and C Programming

1. Discussions on Q&A questions

Q1: Knowing the starting address of arr[], where arr[] is an array of size M. How to determine the memory address of the A-th data point, say A ≤ M?

To determine the memory address of the A-th data point in an array arr[], the following formula can be used: Address of A-th data point = Base Address of arr[] + (A-1) * Size of each element. This formula calculates the offset by multiplying the index minus one (since array indices start at 0) by the size of each element, then adding this offset to the base address of the array.

Q2: Compile the "EE2028_Assign1" project and execute the program. Comment the PUSH {R14} and POP {R14} lines in asm_func(), recompile and execute the program again. Describe what you observe in (i) before and (ii) after removing the two lines and explain why there is a difference. Explain the possible disadvantage(s) when using PUSH and POP instructions.

(i) Before: The function executes correctly with proper return from the function call. (ii) After: The function exhibits unexpected behaviour of crash due to an infinite loop because the return address in the Link Register is not preserved. These instructions, PUSH {R14} and POP {R14}, are important for saving and restoring the state of the Link Register (LR) during function calls, ensuring that the execution flow can correctly return to the point after the function call.

(ii) Disadvantages of using PUSH and POP:

PUSH and POP operations need to access memory, hence additional cycles are required, increasing execution time compared to register operations. Secondly, using PUSH and POP instructions increases the use of stack memory, which is limited in embedded systems, leading to a stack overflow in extreme resource-constrained cases.

Q3: What can you do if you have used up all the general-purpose registers and you need to store some more values during processing?

When all general-purpose registers are used, we can utilise the stack to store values temporarily. By pushing data onto the stack using the PUSH instruction, we can free up registers for other uses. The values can be retrieved later using the POP instruction.

Alternatively, we can use special-purpose registers. Depending on the architecture and context, we might be able to use certain special-purpose registers for temporary storage, though this should be done carefully.

Q4: Consider the following C code snippet and the corresponding assembly routine. After executing the highlighted line (Ln 36, BX LR), which instruction is the Link Register (LR) pointing to at this moment?

After executing the BX LR instruction on line 36 of the assembly routine, the Link Register (LR) is set to point back to the instruction in the main() function immediately following the call to foo(). Since the foo() function is invoked from line 35 in the C code, LR points to the next line of execution in main(), which corresponds to line 36. This line contains the printf statement that outputs the result. Thus, LR points the program flow to execute the printf("Result: %d\n", result); statement upon returning from foo().

Q5. Consider the following C code snippet: Explain what the code is intended to do, focusing on the if-else structure. What is the output of the code? Discuss how the data type of result affects the output when changing the values of a and b. What would happen if result were declared as an integer?

The code compares two integer variables *a* and *b* and checks if a is less than b. If the statement is true, the variable *result* stores the answer of *a* divided by 2.0. If the statement is false, the else block is executed, where *result* stores the answer of *b* divided by 2.0. Given the variable *result* is a floating point variable, the division will retain the decimal value with the correct precision. Since $a = 5$ and $b = 10$, the output will be *result*: *2.5*

If the *result* were declared as an integer, the division would still happen with floating-point precision because of the *2.0*. When assigning the result of the division to an integer variable, however, any decimal part would be truncated. So *result = a / 2.0 ,* where $a = 5$ would give *result: 2.* This means any decimal portion of the answer will be lost.

2. Usage of Registers

The ARM Cortex-M4 architecture offers various registers, specifically utilising:

- **R0-R3**: General-purpose registers, crucial for interfacing between C and assembly functions, facilitating data passing and function returns.
- **R0**: Points to the start of the coefficient array [a,b,c], effectively bridging data from the C program to the assembly routine.
- **R1**: Initially loaded with x_0, the starting point for the gradient descent, which is updated in each iteration to approach the minimum.
- **R2**: Holds the learning rate λ value, adjusted for integer arithmetic to streamline computations by avoiding floating-point operations.
- **R3**: Utilised dynamically within each iteration to compute the gradient $f'(x)=2ax+b$, which is central to determining the direction and magnitude of the next step.
- **R4-R6**: These registers temporarily store the coefficients a, b, and c for quick access during calculations, minimising memory fetch cycles.
- **R7**: Acts as a counter for the number of iterations executed, providing a direct measure of convergence speed and efficiency.
- **R14 (Link Register)**: Holds the return address, ensuring that upon completion of the assembly routine, control is appropriately handed back to the calling function in the C program.

3. Assembly Program Logic

1. Setup and Initialization

- **Register Preservation:** The program starts by pushing `R14` (Link Register) onto the stack, ensuring that the return address is saved and can be restored upon exiting the function. This preserves the calling function's context.
- **Initial Setup:** Registers are initialised with crucial values: `R4` is set to zero to act as a round counter, indicating the number of iterations performed.

2. Subroutine for Coefficient Setup

- **Coefficient Loading:** The subroutine loads coefficient 'b' into `R5` and scales it by 10. This pre-loading and scaling optimise memory access by keeping frequently used data in registers, reducing the overhead of multiple memory accesses during the loop.
- **Scaling Factor Setup:** The scaling factor is also prepared by setting `R6` to 100, preparing it for use in the main computational loop to adjust the gradient calculation by a predefined scaling factor.

3. Gradient Calculation and Application

- **Computational Efficiency:** The loop begins with a simple counter increment (`ADD R4, R4, #1`) to track iterations. It then proceeds to load and compute values directly affecting the optimization:
    - **Loading and Multiplication:** `R3` loads the coefficient 'a', and the multiplication `MUL R3, R3, R1` computes 'a', doubled using `ADD R3, R3, R3` instead of a multiplication by 2, exploiting the faster `ADD` operation.
    - **Gradient Assembly:** The term `2ax` is then added to `b*10` (preloaded into `R5`), and the entire expression is adjusted by the lambda term (`R2`) and scaled, followed by negation and signed division to apply the learning rate.
- **Update Mechanism:** The computed adjustment is applied to `x0` (`R1`), and the updated value is stored back after each iteration, minimising memory access by keeping the iterative variable in a register.

4. Convergence Checking and Loop Control

- **Convergence Testing:** After updating `x`, the change (`R3`) is checked against zero to determine if further iterations are needed. This is a simple yet effective way to check for convergence, ensuring the loop exits when adjustments become negligible.
- **Branching for Iteration:** The `BNE loop` instruction utilises conditional branching to efficiently manage the loop, only continuing if there has been a significant change in the optimization variable.

5. Termination and Cleanup

- **Register Restoration and Exit:** Once the loop concludes (either through convergence or after completing all iterations), the function restores the link register (`R14`) from the stack and returns to the caller, ensuring that all modified registers are reset to their original states to prevent side effects.

6. Memory Management

- **Result Storage:** The final value of `x` and the iteration count are stored in a designated memory location (`ANS`), which is prepared at the beginning. This allows external routines to access the results of the optimization.

4. Optimization and Improvements

4.1 Efficient Register Usage

Strategic register management is crucial in maximising computational speed and minimising reliance on slower memory accesses. In our assembly routine:By keeping temporary variables and frequently accessed data in registers (e.g., coefficients a, b, and intermediate results), we avoid costly memory reads and writes. This approach is especially beneficial in tight loops where the same variables are accessed repeatedly. Specific registers are dedicated to specific tasks, which minimises the need for moving data between registers and ensures smooth execution. For instance, R0-R3 are used for passing arguments and returning results, while R4-R7 hold intermediate values and constants crucial for calculations, reducing the load/store operations to the stack.

4.2 Integer Arithmetic for Efficiency

Integer arithmetic enhances performance by avoiding the computational overhead associated with floating-point operations, which are more resource-intensive and slower on systems without dedicated floating-point hardware. We use scaled integer values to represent potentially fractional data, adjusting calculations to maintain precision without the need for floating-point arithmetic. This is critical in embedded applications on the ARM Cortex-M4, which might not always have an FPU (Floating Point Unit), making integer operations preferable. In gradient calculations, parameters are scaled and managed as integers, and only converted to their final form when necessary, thus preserving speed and resource efficiency.

4.3 Branching and Loop Control

Efficient control flow using conditional branching helps to minimise the execution time and reduces power consumption by eliminating unnecessary iterations. We use instructions like `BNE` (Branch if Not Equal) to control loop execution. This approach ensures that the loop only continues if the criteria are not met, optimising the number of iterations and directly impacting runtime efficiency. The loop checks for convergence in each iteration and breaks out when the minimum is sufficiently approximated. This reduces the computational load by preventing the execution of further unnecessary operations. In the optimization loop, after

each update of the variable x, a convergence check is performed. If the update does not significantly change x, the loop exits early, saving cycles and energy, which is crucial for battery-operated embedded devices.

5. Microarchitecture design

The Microarchitecture diagrams have been modified to support MUL and MLA instructions.

1. A hardware multiplier block has been added.
2. New control signals: MultiControl and M have been included.
3. Modified ALUControl signal.
4. Decoder updated to accept M as input for multiplication operations.
5. Register file expanded to include A4 input and RD4 output.