**CG2028 Assignment 1**

|  |  |
| --- | --- |
| **Name** | **Student Number** |
| Su Menghang, Vincent | A0272102X |
| Chan Xu Ming Ethan | A0273774R |

# Question 1:

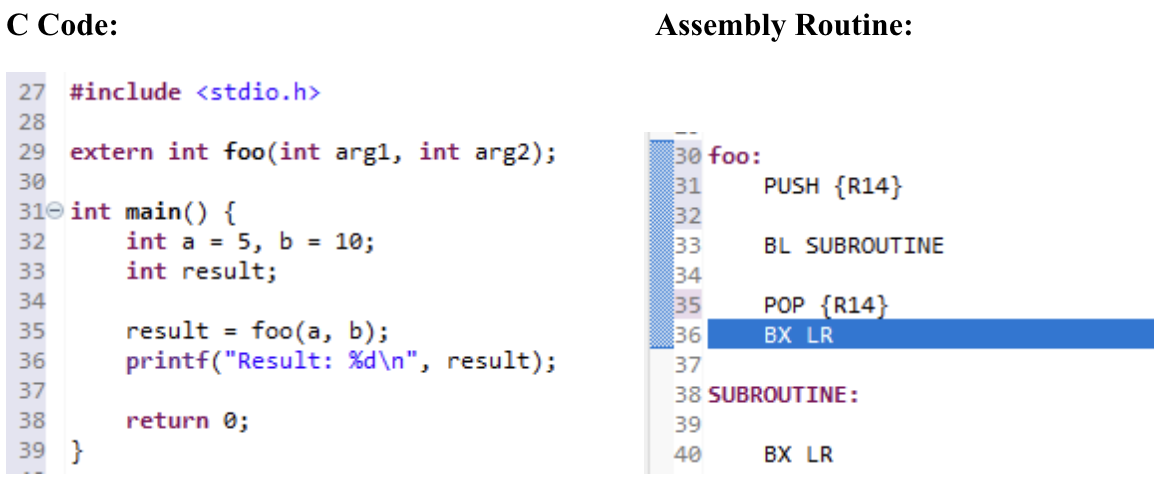
# Knowing the starting address of a 2-d array Arr[][], how to calculate the memory address of element  Arr[A][B] with index A and B starting from 0? Use an equation to explain your answer. (2 mark)

Assuming that this array stores integers, which are 4 bytes in an ARM Cortex M system. Let the number of elements in one row be N.

Therefore, the memory address of the element Arr[A][B] is

# Question 2:

# Consider the following C code snippet and the corresponding assembly routine:



# Read the above code. Assume each line of the C code corresponds to a single line of assembly instruction. After executing the highlighted line (Ln 36, BX LR), which instruction is the Link Register (LR) pointing to at this moment? (2 mark)

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, so BX LR correctly returns to the C code.

# Question 3:

# Describe what you observe in (i) and (ii) and explain why there is a difference. (4 marks)

(i) Comment the **PUSH {R14}** and **POP {R14}** lines in **iir.s,** compile the “Assign1” project and execute the program.

(ii) Uncomment the **PUSH {R14}** and **POP {R14}** lines in **iir.s** recompile and execute the program again.

(i) Without PUSH and POP}, the original LR (which pointed to the printf in Line 36 of C) gets overwritten by the inner BL SUBROUTINE. At the end, BX LR branches back into the instruction itself instead of the C code, causing an infinite loop / incorrect return.

(ii) With PUSH and POP, the original LR is saved on the stack and restored after the subroutine call. Thus BX LR correctly returns execution to the C code at Line 36, and the program behaves as expected.

# Question 4:

# What can you do if you have used up all the general purpose registers and you need to store some more values during processing? (2 marks)

If the amount of values we need to store is more than the registers, we can store the excess and less frequently used variables in the memory. Using the STR instruction, data can be stored in the memory, and using the LDR instruction, we can load it back into the registers for data processing when needed.

# Question 5:

# After finishing writing your code, write down the machine code corresponding to 5 assembly language instructions in your report. You will select the lines of instructions by yourself. You need to select at least 1 data processing instruction, 1 memory accessing instruction, and 1 branching instruction. (5 marks)

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Instruction** | **Binary** | **Hex** |
| **1** | *ADD R12, R12, R6* | *0b0000 0000 1000 1100 1100 0000 0000 0110* | *0x008CC006* |
| **2** | *LDR R4, [R1]* | *0b0000 0101 0001 0001 0100 0000 0000 0000* | *0x05114000* |
| **3** | *BLT EXIT* | *0b1011 1000 0000 0000 0000 0000 0000 1100* | *0xB800000C* |
| **4** | *MUL R6, R6, R8* | *0b0000 0000 0000 0000 0110 1000 0000 0110* | *0x00006806* |
| **5** | STR R4, [R5] | *0b0000 0101 0000 0101 0100 0000 0000 0000* | 0x05054000 |

# Question 6:

# Provide a design showing how the microarchitecture (datapath and control unit) covered in Lecture 4 can be modified to support MLA and MUL instructions. You can assume that a hardware multiplier block is available, which takes in two 32-bit inputs Mult\_In\_A and Mult\_In\_B, and provides a 32-bit output, Mult\_Out\_Product combinationally (i.e., without waiting for a clock edge). You can directly edit the microarchitecture from Lecture 4, Page 28 by taking a screenshot. (6 marks)

A diagram of a computer

AI-generated content may be incorrect.



# Program Logic:

**In SUBROUTINE:**

* *R4 to R12 are pushed onto the stack to save their contents.*
* *b[0] and a[0] are loaded; the result of x\_n \* b[0] / a[0**] is saved into R12.*
* *Loop counter k is initialized to N-1.*
* *Index i (iteration counter) and j (coefficient index) are set up.*

**In LOOP:**

* *Compare k with 0; exit if less than zero.*
* *Ensure i-1 >= 0 to avoid illegal memory access.*
* *Load x\_array[i-1], y\_array[i-1], b[j], a[j].*
* *Compute x\_b = b[j] \* x\_array[i-1], y\_a = a[j] \* y\_array[i-1].*
* *Subtract and divide by a[0], add to running sum in R12.*
* *Update counters: decrement k and i, increment j.*

**In EXIT:**

* *Move result from R12 to R0 (return value).*
* *Apply scaling by dividing by 100.*
* *Store new x\_n and y\_n into arrays at index i.*
* *Increment i and store back.*
* *Restore registers with POP.*
* *Return with BX LR.*

**Improvements Made:**

* In the original C code, inserting a new x\_n and y\_n required \*shifting all elements\* in the arrays, which is inefficient and had a time complexity of O[2n]
* In our assembly, the counter i is used to track the latest index.
* The newest values are stored directly at x\_store[i] and y\_store[i], avoiding shifts.
* This improves efficiency since only a single write is needed per iteration instead of O(2N) shifts, reducing the time complexity to O(1).

**Appendix**

**Contributions:**

|  |  |
| --- | --- |
| **Name** | **Contribution** |
| Vincent | Contributed to both the assembly code and report with a greater focus on the code |
| Ethan | Contributed to both the assembly code and report with a greater focus on the report |