1. Background Concepts

A smart parking system is implemented in a multi-story car park. The car park has F floors and S parking sections on each floor. The system tracks the cars parked in the parking sections in real-time. At the start of each day, the system records the number of cars parked on each floor. Throughout the day, cars enter the car park from time to time. The cars only exit the car park at the end of a day. The objective of this assignment is to develop a system that updates the cars parked at the end of the day, considering the initial state, cars entering, and cars exiting.

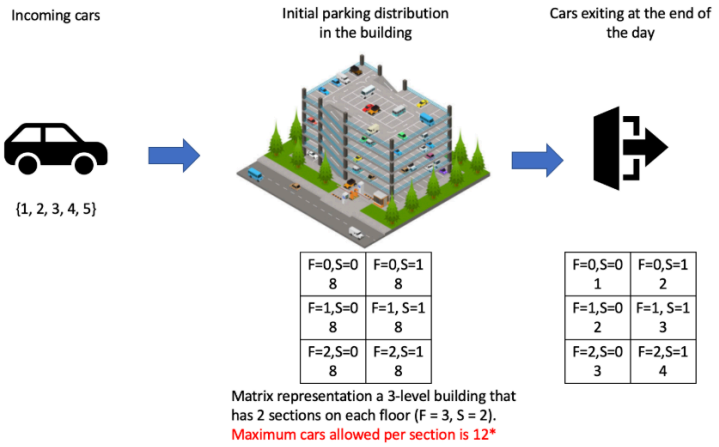

Figure 1 demonstrates an example with a car park having 3 floors and 2 parking sections per floor. The initial state of the car park is represented by a 2D array `{{8, 8}, {8, 8}, {8, 8}}`, where each element represents the number of cars parked in a section. The number of cars entering the car park throughout the day is given as a 1D array `{1, 2, 3, 4, 5}` representing the number of cars entering at different times of the day. The number of cars exiting in each section at the end of the day is represented by `{{1, 2}, {2, 3}, {3, 4}}`.

To update the cars parked, two rules need to be followed:

(1) The maximum number of cars that can park in each section is **12**.

(2) Cars entering the car park are assigned to sections floor by floor, starting from the first section on the first floor (building[0][0]). When a section reaches its maximum capacity of 12, incoming cars are directed to the next section on the same floor.

Therefore, with the given numbers for this example and `SECTION_MAX = 12`, the expected number of cars parked at the end of the day will be `{{11, 10}, {10, 8}, {5, 4}}`.

2. Objective

The objective of this assignment is to develop an ARMv7-M assembly language function **asm_fun()** that updates the cars parked in each section at the end of the day, considering the initial state, cars entering, and cars exiting.

3. Getting Started

Import the `Assign1.zip` archive file which contains the "Assign1" project. Within the `src` folder of this project, there are 2 files you need to pay attention to:

- `asm_func.s`: which is where you will write the assembly language function.
- `main.c`: which is a C program that calls your assembly function.

The C program `main.c` defines a 2D array "building" representing the initial state of the car park, and a 2D array "exit" representing the number of cars exiting each section at the end of the day. It also defines a 1D array "entry" representing the number of cars entering the car park at different times.

The elements of the "building" array are stored row by row in consecutive words in memory. The "exit" and "entry" arrays are stored similarly.

The C program calls the function `asm_fun()` which you will write in assembly language. This function computes the final cars parked and stores it in the "result" array.

Question 1: Knowing the starting address of array `Building[][]`, how to calculate the memory address of element `building[A][B]` with floor index `A` and section index `B`, with the index starting from 0? Use drawing or equation to explain your answer. (1 marks)

The function definition for `asm_func()` in `main.c` is `extern void asm_func(int* arg1, int* arg2, int* arg3, int* arg4)`, which accepts four parameters:

- `building[][]` (Initial state of the car park)
- `exit[][]` (Number of cars exiting each section)
- `entry[]` (Number of cars entering the car park. You can assume the size of this array is always 5.)
- `result[][]` (Array to store the final cars parked, also containing F, S)

4. Parameter passing between C program and assembly language function

In general, parameters can be passed between a C program and an assembly language function through the ARM Cortex-M4 registers. In the function **extern int asm_func (arg1, arg2, ...)**:

arg1 will be passed to the assembly language function asm_func() in the **R0** register, **arg2** will be passed in the **R1** register, and so on. Totally 4 parameters can be passed from C program to assembly program in this way. To **return** value from an assembly language function back to C program, **R0** register is used.

The final cars parked in the building should be stored in the *result[ ][ ]* array so that they can be used to replace the old *building[ ][ ]* array and print out in C program.

In the **asm_func.s**, there is a subroutine called **SUBROUTINE** declared after the main part of assembly program, in order to demonstrate the way to "create and call a function" in an assembly language program. Note: The purpose of this declaration is to lead to the thinking of link register (LR/R14). At the completion of this assignment, **SUBROUTINE** is not compulsory to be used, i.e. you may not have BL SUBROUTINE in your function **asm_fun()**.

Question 2: Consider the following C code snippet and the corresponding assembly routine:

C Code:

```c
#include <stdio.h>

extern int foo(int arg1, int arg2);

int main() {
    int a = 5, b = 10;
    int result;

    result = foo(a, b);
    printf("Result: %d\n", result);

    return 0;
}
```

Assembly Routine:

```
foo:
    PUSH {R14}

    BL SUBROUTINE

    POP {R14}
    BX LR

SUBROUTINE:

    BX LR
```

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? (1 marks)

5. PUSH and POP

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

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

Question 3: Describe what you observe in (i) and (ii) and explain why there is a difference. (2 marks)

6. Programming Tips

Write the code for the assembly language function **asm_func()** to fulfill the objective mentioned in Section 2 after reading the following aspects carefully.

• It is a good practice to push the contents in the affected general purpose registers onto the stack prior to or upon entry into the assembly language function or subroutine, and to pop those values at the end of the assembly language function or subroutine to recover them.

• If you are using a subroutine, special care must be taken with the Link Register (R14). If the content of this register is lost, the program will not be able to return correctly from the subroutine to the calling part of the program.

• If a set of actions are conditional, you may use the ARM assembly language IF-THEN (IT) block feature or conditional branch.

• In a RISC processor such as the ARM, arithmetic and logical operations only operate on values in registers. However, there are only a limited number of general purpose registers, and programs, e.g. complex mathematical functions, may have many terms.

Hint: Use and re-use the registers in a systematic way. Maintain a data dictionary or table to help you keep track of the storage of different terms in different registers at different times.

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? (1 marks)

Note: Each team member should make both joint and specific individual contributions towards the results of this assignment. Verify the correctness of the results computed by the **asm_func()** function you have written by comparing what appears at the console window of the STM32CubeIDE with the desired output shown below:

F=0,S=0 11	F=0,S=1 10
F=1,S=0 10	F=1, S=1 8
F=2,S=0 5	F=2,S=1 4

In the template program, since the displays are achieved by a nested for-loop which starts from index[0][0], you should see the following sequence and order in the console window:

```
RESULT
11      10
10      8
5       4
```

7. C Programming

While you do not need to write in C language for your assignment 1, understanding C language is crucial for you to understand the main.c file in the template.

Question 5: Consider the following C code snippet:

```c
#include <stdio.h>

int main() {
    int a = 5, b = 10;
    float result;

    if (a < b) {
        result = a / 2.0;
    } else {
        result = b / 2.0;
    }

    printf("Result: %.1f\n", result);
    return 0;
}
```

1. Explain what the code is intended to do, focusing on the if-else structure. What is the output of the code? (1 mark)
2. 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? You might need to make appropriate changes to the printf() function call. (1 mark)

8. Test Cases

A total of 8 test cases will be tested on your code, including 5 open test cases and 3 hidden test cases. The open test cases are shown below:

Test case 1

int building[F][S] = {{8,8},{8,8},{8,8}};

int entry[5] = {1,2,3,4,5};

int exit[F][S] = {{1,2},{2,3},{3,4}};

Test case 2

int building[F][S] = {{1,2},{3,4},{5,6}};

int entry[5] = {1,1,1,1,1};

int exit[F][S] = {{1,0},{0,0},{0,0}};

Test case 3

int building[F][S] = {{12,12},{10,5},{3,7}};

int entry[5] = {1,1,1,1,5};

int exit[F][S] = {{1,2},{3,4}, {3,6}};

Test case 4

int building[F][S] = {{12,12},{12,12},{12,12}};

int entry[5] = {0,0,0,0,0};

int exit[F][S] = {{2,2},{3,3},{4,4}};

Test case 5

int building[F][S] = {{9,10},{7,8},{4,4}};

int entry[5] = {2,4,6,8,10};

int exit[F][S] = {{1,1},{1,1},{1,1}};