Write the `depthFirstSearch` subroutine in ARM assembly to generate a maze according to the randomized depth-first search algorithm. The maze will be stored in a NUM_ROW * NUM_COL matrix of bytes with the following encoding for the inner cells (i.e., cells not in the border):

- Bit 0: 1 if the cell has been already visited (i.e., processed) by the algorithm, 0 otherwise
- Bit 1: 1 if there is a passage between the cell and the neighbor at its right, 0 if there is a wall
- Bit 2: 1 if there is a passage between the cell and the neighbor at its bottom, 0 if there is wall
- Bit 3: 1 if there is a passage between the cell and the neighbor at its left, 0 if there is a wall
- Bit 4: 1 if there is a passage between the cell and the neighbor at its top, 0 if there is a wall
- Bit 5, 6, and 7: they are fixed at 0

Instead, the cells along the border have all bits set to 1 (they are not processed by the algorithm).

For example, the initial matrix with NUM_ROW = 6 and NUM_COL = 5 is:

11111111	11111111	11111111	11111111	11111111
11111111	0	1	0	11111111
11111111	0	0	0	11111111
11111111	0	0	0	11111111
11111111	0	0	0	11111111
11111111	11111111	11111111	11111111	11111111

0	1	2	3	4
5	6	7	8	9
A	B	C	D	E
F	10	11	12	13
14	15	16	17	18
19	1A	1B	1C	1D

In the table, the lines between each cell represent a wall (i.e., no passage exists between each couple of neighboring cells). The cell with value 1 is the entrance of the maze. For the sake of clarity, the hexadecimal numeration of the cells according to the row-major order is reported on the right.

The `depthFirstSearch` subroutine receives the following parameters (in the order indicated):

- address of the maze matrix
- number of rows NUM_ROW
- number of columns NUM_COL
- index of the cell at the starting point (in the example it is 7)

The algorithm for the maze generation finds all the neighbors of the current cell not yet visited:

- If there is at least one unvisited neighbor, a passage between the current cell and a chosen neighbor is created (changing the corresponding bits in both cells). The index of the current cell is pushed on the stack. The neighbor is marked as visited (by setting its bit 0) and becomes the current cell for the next iteration of the algorithm.
- If there are not unvisited neighbors, the content of the current cell does not change. Pop a value from the stack: it will become the current cell for the next iteration of the algorithm.

The algorithm ends when the stack is empty (meaning that all cells have been visited).

In details, you have to implement the search of visited or unvisited neighbors in the following way:

- Set r0 = 0 if the neighboring cell at the right has not been visited, 1 otherwise
- Set r1 = 0 if the neighboring cell at the bottom has not been visited, 1 otherwise
- Set r2 = 0 if the neighboring cell at the left has not been visited, 1 otherwise
- Set r3 = 0 if the neighboring cell at the top has not been visited, 1 otherwise

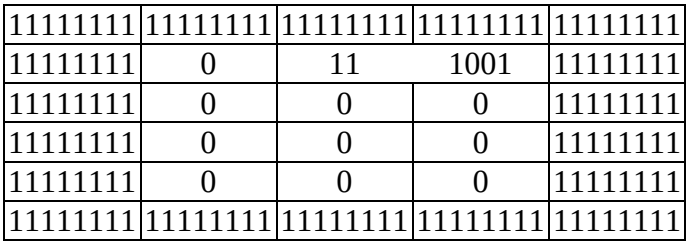
Then, `depthFirstSearch` calls the `chooseNeighbor` subroutine, which returns

- 1 if r0 = 0
- 2 if r0 = 1 and r1 = 0
- 3 if r0 = r1 = 1 and r2 = 0
- 4 if r0 = r1 = r2 = 1 and r3 = 0
- 0 if r0 = r1 = r2 = r3 = 1 (meaning that all neighbors have been visited)

For example, if `chooseNeighbor` returns 2, `depthFirstSearch` creates a passage between the current cell and its neighbor at the bottom: bit 2 of current cell and bit 4 of the neighbor are set. Example of execution with the 6x5 matrix shown at the beginning.

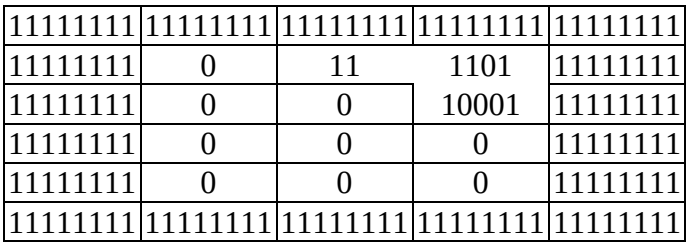
Iteration 1

Index of current cell is 7.
r0 = 0, r1 = 0, r2 = 0, r3 = 1.
`chooseNeighbor` returns 1.
A passage is carved between cells 7 and 8.
The new current cell is 8.
Stack now contains 7.



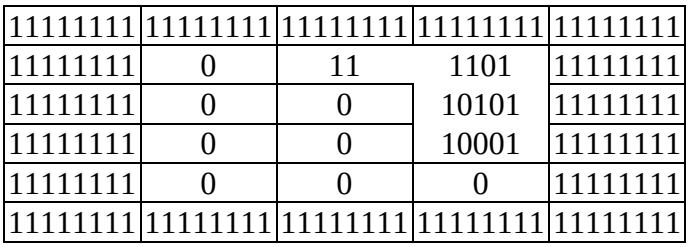
Iteration 2

Index of current cell is 8.
r0 = 1, r1 = 0, r2 = 0, r3 = 1.
`chooseNeighbor` returns 2.
A passage is carved between cells 8 and D.
The new current cell is D.
Stack now contains 8, 7.



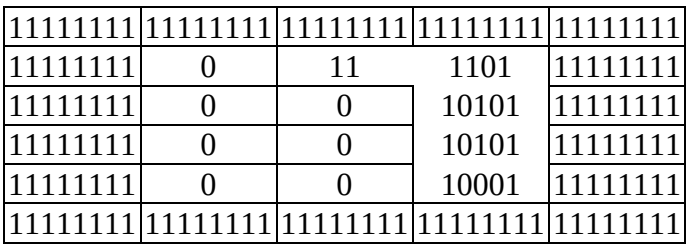
Iteration 3

Index of current cell is D.
r0 = 1, r1 = 0, r2 = 0, r3 = 1.
`chooseNeighbor` returns 2.
A passage is carved between cells D and 12.
The new current cell is 12.
Stack now contains D, 8, 7.



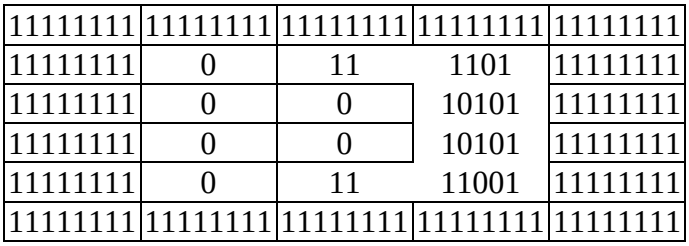
Iteration 4

Index of current cell is 12.
r0 = 1, r1 = 0, r2 = 0, r3 = 1.
`chooseNeighbor` returns 2.
A passage is carved between cells 12 and 17.
The new current cell is 17.
Stack now contains 12, D, 8, 7.



Iteration 5

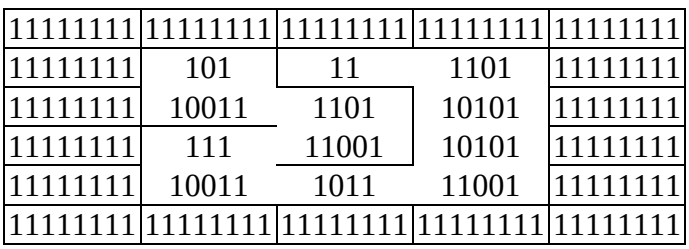
Index of current cell is 17.
r0 = 1, r1 = 1, r2 = 0, r3 = 1.
`chooseNeighbor` returns 3.
A passage is carved between cells 17 and 16.
The new current cell is 16.
Stack now contains 17, 12, D, 8, 7.



…

Iteration 11

After iteration 11, the current cell is 6.
Stack contains B, C, 11, 10, 15, 16, 17, 12, D, 8, 7.
Other 11 iterations follow, popping a value from the stack until the stack is empty, because no neighbors remain.



Important notes:

1. Create a new project with Keil inside the “ARM” directory and write your code there. The “ARM” directory contains some subdirectories that you can add to your project if you need them. **It also contains the startup_LPC17xx.s file with the Reset_Handler procedure and the declaration of the memory areas.**
2. The assembly subroutine must comply with the ARM Architecture Procedure Call Standard (AAPCS) standard (in terms of parameter passing, returned value, callee-saved registers).
3. Click on the following links to open web pages with the ARM instruction set

The maze generated in the previous exercise is always the same because the `chooseNeighbor` subroutine is deterministic. We want to write a `chooseRandomNeighbor` subroutine that chooses a random neighbor exploiting the systick timer. First, in the Reset_Handler, set the Reload Value Register of the systick timer to 0xFFFFF and configure the systick timer in order not to generate an interrupt when the timer counter reaches 0. Then, start the timer.

The `chooseRandomNeighbor` subroutine receives the same parameters as `chooseNeighbor`

- r0 = 0 if the neighboring cell at the right has not been visited, 1 otherwise
- r1 = 0 if the neighboring cell at the bottom has not been visited, 1 otherwise
- r2 = 0 if the neighboring cell at the left has not been visited, 1 otherwise
- r3 = 0 if the neighboring cell at the top has not been visited, 1 otherwise

First, the subroutine pushes some values on the stack:

- if r0 = 0, it pushes 1
- if r1 = 0, it pushes 2
- if r2 = 0, it pushes 3
- if r3 = 0, it pushes 4

If no values are pushed, the subroutine immediately returns 0. Otherwise, it counts how many registers, among the input parameters, are 0. Then, it computes the module between the systick timer counter and the number of registers equal to 0. For example, if r0 = 0, r1 = 0, r2 = 0, r3 = 1, it computes the module with 3. The module indicates the index of the element of the stack to retrieve, which is the return value. It is important to pop all elements from the stack before concluding the subroutine to leave the stack balanced.

Note about the systick timer.
The systick timer is configured by means of the following registers:

- Control and Status Register: size 32 bits, address 0xE000E010
- Reload Value Register: size 24 bits, address 0xE000E014
- Current Value Register: 24 bits, address 0xE000E018

In the provided startup_LPC17xx.s file, the addresses are defined as constants.

The meaning of the bits in the Control and Status Register is as follows:

- Bit 16 (read-only): it is read as 1 if the counter reaches 0 since last time this register is read; it is cleared to 0 when read or when the current counter value is cleared
- Bit 2 (read/write): if 1, the processor free running clock is used; if 0, an external reference clock is use
- Bit 1 (read/write): if 1, an interrupt is generated when the timer reaches 0; if 0, the interrupt is not generated
- Bit 0 (read/write): if 1, the systick timer is enabled; if 0, the systick timer is disabled.

The Reload Value Register stores the value to reload when the timer reaches 0.
The Current Value Register stores the current value of the timer. Writing any number clears its content