INTRODUCTION TO COMPUTERS II
LAB 5: MATRICES

The elements of a matrix are stored in memory ordered by rows, that is, from left to right and from top to bottom in consecutive memory addresses. Thus, a matrix M×N is stored as if it were a vector of M·N components, where the first m elements correspond to the first row of the matrix, the next m elements to the second row, and so on. For example, the following 3×4 matrix is stored in memory as a vector of 12 components.

0	1	2	3
4	5	6	7
8	9	10	11

```
m: .word 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
```

The following assembly fragment loads the element m[1][1] of the matrix.

```
la s1, m          //s1 holds the base address
lw a0, 20(s1)     //s1 + 20 gives the effective address
                  //a0 now contains a 5
```

To begin, develop a function in RISC-V assembly that copies a square matrix into another according to the following algorithm.

```
void matrixCopy(int n, int x[n][n], int z[n][n]) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            z[i][j] = x[i][j];
}
```

Next, develop a function in RISC-V assembly that multiplies two square matrices of the same dimension according to the following algorithm. In this function, use the **mul** instruction to multiply values.

```
void matrixMul(int n, int x[n][n], int y[n][n], int z[n][n]) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            z[i][j] = 0;
            for (int k = 0; k < n; k++)
                z[i][j] = z[i][j] + x[i][k] * y[k][j];
        }
}
```

Finally, develop a function in RISC-V assembly that calculates the power **e** of a square matrix **x** according to the following algorithm. The space for the auxiliary matrix **aux** (since it is a local variable) should be created on the stack above the saved registers and should be removed before restoring them. It is not necessary to use the **fp** register; using **sp** as the base register for **aux** is sufficient.

```
void matrixPow(int n, int x[n][n], int e, int z[n][n]) {
    int aux[n][n];

   for (int j=0; j<n ; j++) //initializes z with the identity matrix
       for (int k=0; k<n ; k++)
           if (j==k)
               z[j][k] = 1;
           else
               z[j][k] = 0;

    for (int i = 1; i <= e; i++) {
        matrixMul(n, x, z, aux);
        matrixCopy(n, aux, z);
    }
}
```

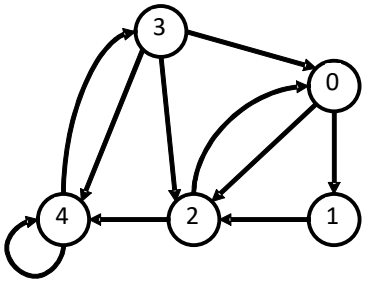
To test the correct functionality of the previous functions, use the following C program, which, given a directed graph represented by its adjacency matrix, calculates for a pair of nodes the number of different paths connecting them with exactly a given number of steps (transitions). You do not have to translate this to assembly, use it directly in C.

```
#define N     5     //number of nodes in the graph
#define STEPS 3     //number of steps (transitions) in the path
#define ORG   0     //source node
#define DST   3     //destination node

extern void matrixPow(int n, int [n][n], int, int [n][n]);

int graph[N][N] =   //adjacency matrix of the graph
{
    {0, 1, 1, 0, 0},
    {0, 0, 1, 0, 0},
    {1, 0, 0, 0, 1},
    {1, 0, 1, 0, 1},
    {0, 0, 0, 1, 1}
};
int z[N][N];        //result matrix of the algorithm
int paths;

void main() {

    matrixPow(N, graph, STEPS, z);
    paths = z[ORG][DST];

   while(1);
}
```

In this program, the adjacency matrix **`graph`** represents the following directed graph:

destination node

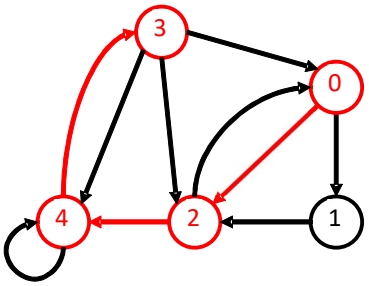
source node	**0**	**1**	**2**	**3**	**4**
0	0	1	1	0	0
1	0	0	1	0	0
2	1	0	0	0	1
3	1	0	1	0	1
4	0	0	0	1	1



Running the program with the given data results in the variable **`paths`** taking the value 1, since there is exactly one path between node 0 and node 3 that passes through exactly 3 transitions. The resulting auxiliary matrix **`z`** (which, for each pair of nodes, calculates the number of different paths with 3 transitions connecting them) is as follows:

destination node

source node	**0**	**1**	**2**	**3**	**4**
0	1	1	1	**1**	2
1	0	1	1	1	1
2	2	0	2	1	3
3	2	1	3	2	4
4	2	1	2	2	4



However, it is recommended that, to simplify the debugging of the assembly functions, you modify the provided C program and test each of them individually with different example matrices.