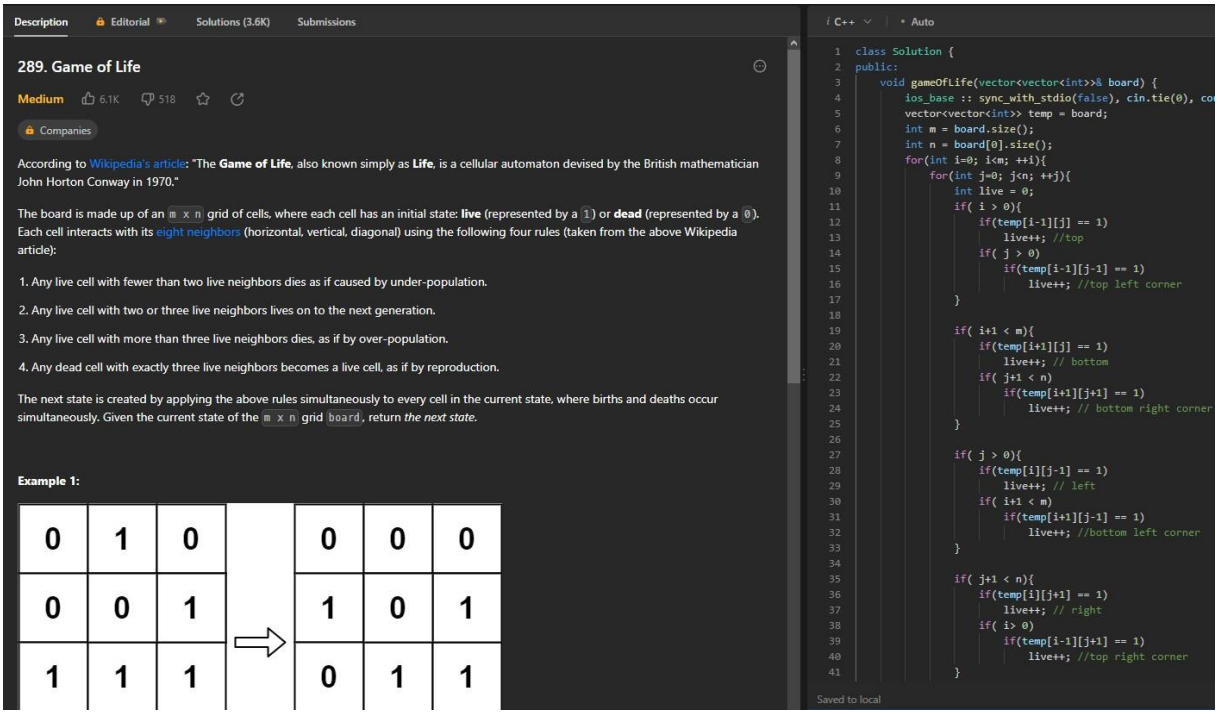
Introduction to Embedded Systems EHB 326E #HW2

Ataberk Demirkaya

040200252 – Game of Life



Description | Editorial | Solutions (3.6K) | Submissions

289. Game of Life

Medium 6.1K 518

Companies

According to Wikipedia's article: "The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

The board is made up of an `m x n` grid of cells, where each cell has an initial state: **live** (represented by a `1`) or **dead** (represented by a `0`). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

1. Any live cell with fewer than two live neighbors dies as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by over-population.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The next state is created by applying the above rules simultaneously to every cell in the current state, where births and deaths occur simultaneously. Given the current state of the `m x n` grid `board`, return *the next state*.

Example 1:

0	1	0	⇨	0	0	0
0	0	1		1	0	1
1	1	1		0	1	1

C++ | Auto

```
class Solution {
public:
    void gameOfLife(vector<vector<int>>& board) {
        ios_base :: sync_with_stdio(false), cin.tie(0), co
        vector<vector<int>> temp = board;
        int m = board.size();
        int n = board[0].size();
        for(int i=0; i<m; ++i){
            for(int j=0; j<n; ++j){
                int live = 0;
                if( i > 0){
                    if(temp[i-1][j] == 1)
                        live++; //top
                    if( j > 0)
                        if(temp[i-1][j-1] == 1)
                            live++; //top left corner
                }

                if( i+1 < m){
                    if(temp[i+1][j] == 1)
                        live++; // bottom
                    if( j+1 < n)
                        if(temp[i+1][j+1] == 1)
                            live++; // bottom right corner
                }

                if( j > 0){
                    if(temp[i][j-1] == 1)
                        live++; // left
                    if( i+1 < m)
                        if(temp[i+1][j-1] == 1)
                            live++; //bottom left corner
                }

                if( j+1 < n){
                    if(temp[i][j+1] == 1)
                        live++; // right
                    if( i> 0)
                        if(temp[i-1][j+1] == 1)
                            live++; //top right corner
                }
```

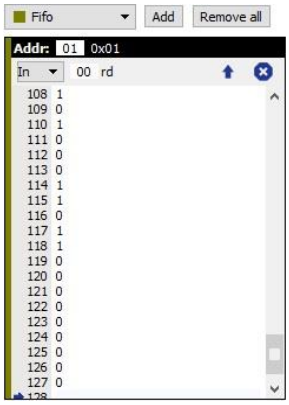
Saved to local

Above, there is the description of the problem that I am supposed to solve using picoblaze and assembly. Basically given a grid, you should find the next state of the input matrix according to the rules.

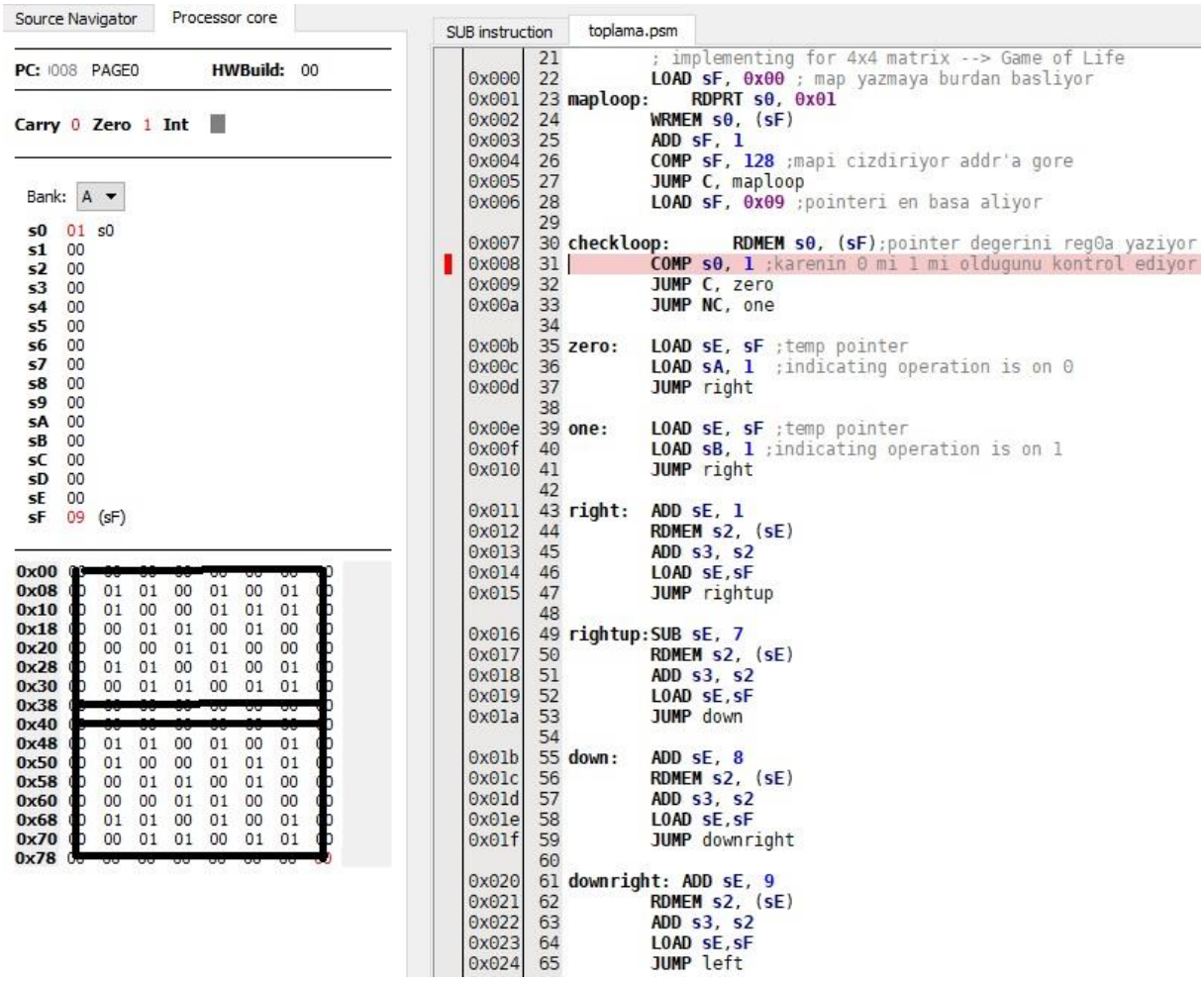
The rules: If a live cell (1) has 2 or 3 neighbors it keeps living, otherwise it dies (0).

If a dead cell (0) has 3 neighbors it revives (1), otherwise keeps its state.

The algorithm in C++ traverses through every square in the grid and checks its 8-way neighbors in order to determine its next state. However, to not cause confusion it copies the existing input grid to a temporary one to implement the algorithm on for printing the next state of the grid. This way the algorithm is not disrupted by previous outputs on previous squares.



This is the input that I apply to the port 0x01 in order to write the grid.



Above you are seeing the initial state of my assembly code for implementing this problem. Since memory is limited with Picoblaze, I chose a 6x6 approach for my grid to implement my algorithm on. In the bottom left you can see the grid printed along with its copy at the bottom. All changes will be done to the bottom one using a temporary pointer “sE” while the upper one will remain as my original input.

I chose register “sF” as a pointer to the grid addresses to write the map onto the board and register “s0” acts as a port reader which reads the data from the input stream from the port 0x01 for keeping track of which number (0 or 1) to write to the grid. By initializing a loop and setting the counter at 128 I wrote the grid onto the scratchpad.

Since I had a 150 line limit due to the policy of Fidex IDE, to keep things simples I left a barriers of zeros outside the grid thus removing the need of separately writing conditions for corners and edges. Having zeros as a neighbor does not affect the rules at all therefore its implementable.

Then the process is pretty simple. In checkloop, it checks whether the current square has a dead cell or a live cell (0 or 1). Then it jumps to the according loop. For both cases I attended sA and sB registers for keeping track of the current data being a zero or a one. This will come in handy when the change loop based on the rules activates because it has different actions for a dead cell and a live cell.

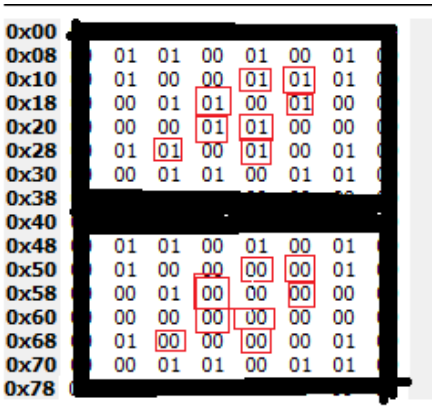
Then the next four loop you see in the upper picture which are named:
”right”,”rightup”,”down”,downup”. They check the neighbors of the current square. Every square has 8 neighbors so we need 8 directions to check. Therefore I wrote 8 loops like that in total, the rest of them you can check in the full code at the bottom. In these directional check functions I used the register “s3” for keeping track of how many live neighbors the current cell has.

```
up:     SUB sE,8
        RDMEM s2, (sE)
        ADD s3, s2
        LOAD sE,sF
        COMP sA, 1 ;hangisinin sonucuna gidecegini seciyor
        JUMP Z, resultzero
        COMP sB, 1
        JUMP Z, resultone

resultzero: COMP s3, 3
            JUMP Z, effectzero
            ADD sF, 1
            LOAD s3, 0
            LOAD sA, 0
            JUMP checkloop

resultone:  COMP s3, 2
            JUMP Z, effectone
            COMP s3, 3
            JUMP Z, effectone
            ADD sE, 64
            WRMEM s9, (sE)
            LOAD sE, 0
            ADD sF, 1
            LOAD s3, 0
            LOAD sB, 0
            JUMP checkloop

effectzero: ADD sE, 64 ;sifirda degisim varsa burda
            LOAD sE, 1
            ADD sF, 1
            LOAD s3, 0
            LOAD sA, 0
            JUMP checkloop

effectone:  ADD sF, 1 ;birde degisim yoksa burda
            LOAD s3, 0
            LOAD sB, 0
            JUMP checkloop
```

In these last 5 functions the outcome for the current cell will be determined and pointer will move to the next address (cell). In the last directional check which is “up”, by using the sA and sB registers the code moves to the outcome of either zero or one. Then in these resulting loops outcomes are determined according to the rules of the game. For example in resultone, if the current cell has 2 or 3 live neighbors it moves to effectone which does not change the current state and moves on to the next cell. However if it does not have 2 or 3 live neighbors, it kills the current cell then moves on to the next one. Of course before moving to the next cell, previously used registers like s3, sA, sB, sE are resetted.

In conclusion by doing the to every cell in the original grid (upper) then implementing the changes in the temporary one (bottom). We acquire the solution to the problem.

The ultimate result and changes are highlighted.



Full Code:

```
        21          ; implementing for 4x4 matrix --> Game of Life
0x000   22          LOAD sF, 0x00 ; map yazmaya burdan basliyor
0x001   23 maploop:     RDPRT s0, 0x01
0x002   24          WRMEM s0, (sF)
0x003   25          ADD sF, 1
0x004   26          COMP sF, 128 ;mapi cizdiriyor addr'a gore
0x005   27          JUMP C, maploop
0x006   28          LOAD sF, 0x09 ;pointeri en basa aliyor
        29
0x007   30 checkloop:       RDMEM s0, (sF);pointer degerini reg0a yaziyor
0x008   31          COMP s0, 1 ;karenin 0 mi 1 mi oldugunu kontrol ediyor
0x009   32          JUMP C, zero
0x00a   33          JUMP NC, one
        34
0x00b   35 zero:    LOAD sE, sF ;temp pointer
0x00c   36          LOAD sA, 1  ;indicating operation is on 0
0x00d   37          JUMP right
        38
0x00e   39 one:     LOAD sE, sF ;temp pointer
0x00f   40          LOAD sB, 1 ;indicating operation is on 1
0x010   41          JUMP right
        42
0x011   43 right:   ADD sE, 1
0x012   44          RDMEM s2, (sE)
0x013   45          ADD s3, s2
0x014   46          LOAD sE,sF
0x015   47          JUMP rightup
        48
0x016   49 rightup:SUB sE, 7
0x017   50          RDMEM s2, (sE)
0x018   51          ADD s3, s2
0x019   52          LOAD sE,sF
0x01a   53          JUMP down
        54
0x01b   55 down:    ADD sE, 8
0x01c   56          RDMEM s2, (sE)
0x01d   57          ADD s3, s2
0x01e   58          LOAD sE,sF
0x01f   59          JUMP downright
        60
0x020   61 downright: ADD sE, 9
0x021   62          RDMEM s2, (sE)
0x022   63          ADD s3, s2
0x023   64          LOAD sE,sF
0x024   65          JUMP left
        66
```

```
0x025  67 left:    SUB sE, 1
0x026  68          RDMEM s2, (sE)
0x027  69          ADD s3, s2
0x028  70          LOAD sE,sF
0x029  71          JUMP downleft
       72
0x02a  73 downleft: ADD sE, 7
0x02b  74          RDMEM s2, (sE)
0x02c  75          ADD s3, s2
0x02d  76          LOAD sE,sF
0x02e  77          JUMP leftup
       78
0x02f  79 leftup:  SUB sE, 9
0x030  80          RDMEM s2, (sE)
0x031  81          ADD s3, s2
0x032  82          LOAD sE,sF
0x033  83          JUMP up
       84
0x034  85 up:      SUB sE,8
0x035  86          RDMEM s2, (sE)
0x036  87          ADD s3, s2
0x037  88          LOAD sE,sF
0x038  89          COMP sA, 1 ;hangisinin sonucuna gidecegini seciyor
0x039  90          JUMP Z, resultzero
0x03a  91          COMP sB, 1
0x03b  92          JUMP Z, resultone
       93
0x03c  94 resultzero: COMP s3, 3
0x03d  95             JUMP Z, effectzero
0x03e  96             ADD sF, 1
0x03f  97             LOAD s3, 0
0x040  98             LOAD sA, 0
0x041  99             JUMP checkloop
      100
0x042 101 resultone:  COMP s3, 2
0x043 102             JUMP Z, effectone
0x044 103             COMP s3, 3
0x045 104             JUMP Z, effectone
0x046 105             ADD sE, 64
0x047 106             WRMEM s9, (sE)
0x048 107             LOAD sE, 0
0x049 108             ADD sF, 1
0x04a 109             LOAD s3, 0
0x04b 110             LOAD sB, 0
0x04c 111             JUMP checkloop
      112
```

```
0x04d 113 effectzero: ADD sE, 64 ;sifirdadegisim varsa burda
0x04e 114             LOAD sE, 1
0x04f 115             ADD sF, 1
0x050 116             LOAD s3, 0
0x051 117             LOAD sA, 0
0x052 118             JUMP checkloop
      119
0x053 120 effectone:  ADD sF, 1 ;birde degisim yoksa burda
0x054 121             LOAD s3, 0
0x055 122             LOAD sB, 0
0x056 123             JUMP checkloop
```