Project: Fun with Cellular Automata and HDMI

Assigned: Monday 4/1; Due **Friday 5/3** (midnight)

Instructor: Nate Lannan

1. Introduction

This project is meant to be an encompassing project that gives you the full experience of all that you learned in this course. It will bring ideas that we covered as well as had within laboratories. It will also try to reinforce all the skills you learned during your time in laboratory.

We will revisit using something completely different but in this setting we will have an idea of what combinational and sequential digital systems are. The project involves something called cellular automata by using the Game of Life invented by John Horton Conway in 1970 [1]. Cellular automata uses discrete, abstract computation systems that are proven useful in numerous complexity models as even more advanced dynamic system over many scientific fields. This field is extremely exciting in that it utilizes computation in some form to model repetitive systems to accomplish an outcome. Much of the ideas on cellular automata are intertwined within new ideas on artificial intelligence and machine learning.

The game works by giving an initial matrix and then the game repeatedly creates multiple matrices after processing the data. Since the game works by itself based on an initial grid matrix, it is sometimes called a zero-player game. The game works by computing a two-dimensional grid of square cells. Currently, the game is only designed for an 8×8 matrix, however, it can easily be expanded to larger sizes. Every cell interacts with its nearest neighbor that is horizontal, vertical, and diagonally adjacent to each cell in the grid (sometimes called a Moore Neighborhood). At each iteration, several rules are utilize based on whether the grid is alive (bit=1) or dead (bit=0). The rules are, as follows [2]:

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 neighbor's lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overcrowding.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

To make the matrix easier to use, the grid is only populated with a 1 or a 0 for alive and dead, respectively. The initial pattern for the matrix is called a seed of the system and it determines the patterns that each iteration produces. Each generation is subsequently computed by the rules given and based on the seed interesting outcomes possibly emerge. For more information, please consult the WikiPedia page at the following URL: `http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`.

2. Background

The implementation uses a simple register to hold the size of the grid space. For example, the default implementation is given for a 8×8 matrix, as shown in Figure 1, therefore, the register size will be allocated for 64 bits where each bit determines whether a specific grid is alive or dead. Larger matrices can easily be created by expanding the register size, however, special attention has to be made when accessing a specific grid point. For example, if a user wants to target (row = 4) and (column = 5) of the matrix, the grid point will be given by: $(4-1)*8+(5-1) = 28$. Each row and column is subtracted by 1, since the starting grid point at the upper left hand corner is $(0,0)$.

An 8×8 grid looks something like Figure 1 where each grid can contain a 0 or 1 indicating that spot is either dead or alive, respectively. Again, the top left-hand portion of the grid is $(0,0)$. Although 8×8 is rather small, the matrix can easily be expanded, if necessary by examining the datpath Verilog code. A sample Conway game of life is shown at the following URL: `http://pmav.eu/stuff/javascript-game-of-life-v3.1.1/`.

As stated previously, the matrix is held in a register based on the number of columns and rows. For example, a register in Table 1 is shown for a 4×4 matrix. The key to using the matrix is making sure that a

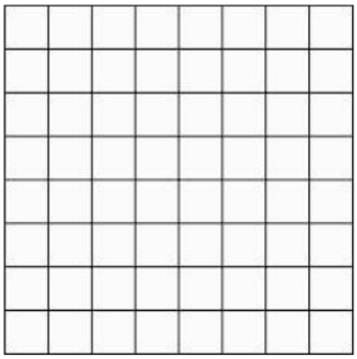

Figure 1: Default 8 x 8 Conway Game of Life Matrix

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	1	1	0	0	1	1	1	1	1	1	0	0	0	0

Table 1: Sample Register for a 4 x 4 Matrix

new matrix (called an evolving matrix) accesses the register based on several points, as expressed previously.

The initial matrix is called the seed matrix and many interesting patterns can be generated based on different seeds. You can search the Internet for some good seeds. I used the following seed to test the design and you are welcome to use the same seed for your testing (given as a Verilog statement) as shown in Figure 2:

The key to digital systems is that digital logic can process data in parallel. To help you with your project, I have coded up the project datapath in Verilog to show how this works. The baseline project involves three simple modifications to complete the project: adding the control logic, adding logic to display to a screen, and adding a LFSR pseudo random sequencer to generate your initial seed. In order to complete this project, the following items must be functional on the board:

- Control logic should be added to control each iteration of the zero-player game.
- A LFSR sequencer to initialize the game. You should be able to view the sequencer change every clock cycle using a button or switch before the seed is loaded and the game is started. A seperate button or switch should be used to load the current “random seed" when you are ready to start the game.
- A video system in order to handle output to a screen through HDMI.
- All testbenches for the system, control, and datapath.
- Cleaned up code removing any unnecessary logic and/or states.
- A report documenting the complete game and its operation.
- Anything you can think of to promote your project!

This project is not difficult and it is an excellent choice for groups that want a straightforward project and/or who wish to spend a minimal amount of time as possible on the project. However, it is in your best

```
assign grid = 64'h0412_6424_0034_3C28;
```

Figure 2: Good Starting Seed for Conway’s Game of Life

interest to make the project better! Any modification beyond the baseline project will incur extra points that could help compensate for a bad test, missing homework, or bad quiz. The following are potential modifications you can easily add for extra credit:

- Increase matrix (originally just 8x8)
- A more complicated LFSR which avoids Lock-up
- Add some cool patterns (e.g., gliders, spaceship – check Internet for good starting seeds)
- Mutations/Variations on Life (add extra rules → sometimes called better CA)
- Make code easier to adapt size (matrix grid)
- 2-dimensional – with color.
- Add poster to explain to masses!
- Create a YouTube® channel on this!
- Add music by attaching speakers (ask TA) to output music something is happening.

3. Implementation

The basic idea for the inputs and output includes your move which you should add through the switches. Before you implement the LFSR you will technically only need a "Start" switch to get things going. After you implement the LFSR you will need a switch to observe the LSFR sequencer, and a switch to load the seed and start the game. The output of the datapath should go to the HDMI section. The controller for the HDMI will be given to you, but you will be required to figure out how to display on it the screen (with help of course).

The main key elements of the design will be provided to you, such as the `evolve` and `rules` modules. But, you will have to add registers, control logic (e.g., Finite State Machines), multiplexors, input/output signals. It is advisable to use as many input/output signals to help you debug what is going on as you simulate and get to your final implementation.

You should use many of the elements discussed in our textbook [3]. I would highly advise using the *idioms* we discussed in class and in the textbook for things like registers making sure that you also reset or enable them appropriately. That is, you should also incorporate a **reset** somewhere into your design. All of the HDL discussed in the textbook [3] is on Canvas as a zip file. Feel free to use this HDL in any way you wish.

HDMI or High-Definitionl Multimedia Interface is one of the most popular interfaces for displaying video. Unfortunately, HDMI is proprietary despite being an open standard which means this project is only for use in information or educational designs and if you wanted to use what we have in this project for further development (e.g., commercial applications), you may need to contact legal help either here on campus or elsewhere. However, I hope you learn something about HDMI from this project related to the interface.

We are not going to use anything spectacular for the HDMI interface - we are only outputing our matrix. You will need to use the attached SystemVerilog example to help you display things properly on the screen. Your repository will have a sample Vivado project that has the HDMI set up correctly. However, you will need to use this project in the Vivado directory along with the sample code to display the output of your game correctly.

The HDMI task will be somewhat challenging in that many of you have never utilized video before. However, video is one of the cooler elements of this project. The key to the video is that you have to make sure your datapath/control completes a matrix computation before it goes to the next display pattern. This means you may need a slower clock to handle this, similar to Lab 3.

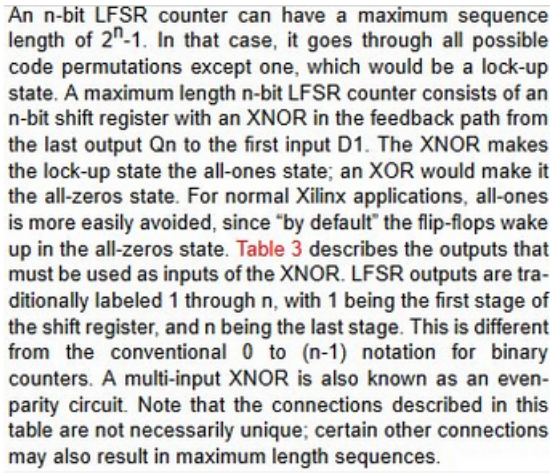
An n-bit LFSR counter can have a maximum sequence length of 2^n-1. In that case, it goes through all possible code permutations except one, which would be a lock-up state. A maximum length n-bit LFSR counter consists of an n-bit shift register with an XNOR in the feedback path from the last output Qn to the first input D1. The XNOR makes the lock-up state the all-ones state; an XOR would make it the all-zeros state. For normal Xilinx applications, all-ones is more easily avoided, since "by default" the flip-flops wake up in the all-zeros state. Table 3 describes the outputs that must be used as inputs of the XNOR. LFSR outputs are traditionally labeled 1 through n, with 1 being the first stage of the shift register, and n being the last stage. This is different from the conventional 0 to (n-1) notation for binary counters. A multi-input XNOR is also known as an even-parity circuit. Note that the connections described in this table are not necessarily unique; certain other connections may also result in maximum length sequences.

Figure 3: Simplified XNOR implementation

3.1 LFSR

A Linear Feedback Shift Register (LFSR) is a shift register where the input is a linear function of the current state of the register. These shift resgisters are often used for fast generation of psuedo random sequences. They are unsophisticated, but have very low overhead and in our case are quite useful to generate a new "random" seed each time we play the game. There are many options for how togenerate the new input into the shift register, we are most interested in Maximal LFSRs. These are LFSRs that cycle through all possible values (except one) before repeating. For instance a 4 bit LFSR will cycle through 15 values before repeating.

We want to have a 64 bit (8 x 8 game of life grid) LSFR cycle through a random sequence at each clock pulse. When your final project is complete you will have a button or switch to display the cycling of the LFSR before the game runs, and a button or switch to load the random seed and start the game.

We will be using a simplified version of an LFSR based on this document: `https://courses.cs.washington.edu/courses/cse369/16wi/labs/xapp052_LFSRs.pdf` There is a lot of implementation in this documentation to avoid a "lock-state" but we will be implementing the XNOR or XOR simplified version as shown in Figure 3.

In this documentation the writers opt for an XNOR implementation to avoid troubles when the LFSR is initialized with all 0s. We do not care bout this issue because an all zero seed is not useful for us. Thus, we are able to use a simple XOR implementation. Once your LFSR is implemented, you should answer the question: "What happens when we initialize the LFSR with all zeros?".

Fortunately the Xilinx documentation gives us a table of "taps" for various LFSR bit lengths (Figure 4), to produce a maximal LSFR for that bit length. These "taps" are the bit numbers that need to be XORed to produce the input into the shift operation.

For example a 16 bit LFSR using the taps specified in the Xilinx document would look like the shift register in Figure 5.

You will need to:

1. Generate an LFSR for a small bit value (8 or 16 bits)
2. Prove that LFSR is maximal through the use of a test bench and the included python program first_repeat.py if desired
3. design a 64 bit LFSR to cycle through a random sequence at each clock pulse to initialize the game.
4. implement a switch to view this cycling process before the seed is loaded and the game begins.

n	XNOR from	n	XNOR from	n	XNOR from	n	XNOR from
3	3,2	45	45,44,42,41	87	87,74	129	129,124
4	4,3	46	46,45,26,25	88	88,87,17,16	130	130,127
5	5,3	47	47,42	89	89,51	131	131,130,84,83
6	6,5	48	48,47,21,20	90	90,89,72,71	132	132,103
7	7,6	49	49,40	91	91,90,8,7	133	133,132,82,81
8	8,6,5,4	50	50,49,24,23	92	92,91,80,79	134	134,77
9	9,5	51	51,50,36,35	93	93,91	135	135,124
10	10,7	52	52,49	94	94,73	136	136,135,11,10
11	11,9	53	53,52,38,37	95	95,84	137	137,116
12	12,6,4,1	54	54,53,18,17	96	96,94,49,47	138	138,137,131,130
13	13,4,3,1	55	55,31	97	97,91	139	139,136,134,131
14	14,5,3,1	56	56,55,35,34	98	98,87	140	140,111
15	15,14	57	57,50	99	99,97,54,52	141	141,140,110,109
16	16,15,13,4	58	58,39	100	100,63	142	142,121
17	17,14	59	59,58,38,37	101	101,100,95,94	143	143,142,123,122
18	18,11	60	60,59	102	102,101,36,35	144	144,143,75,74
19	19,6,2,1	61	61,60,46,45	103	103,94	145	145,93
20	20,17	62	62,61,6,5	104	104,103,94,93	146	146,145,87,86
21	21,19	63	63,62	105	105,89	147	147,146,110,109
22	22,21	64	64,63,61,60	106	106,91	148	148,121

Figure 4: Tap locations for a maximal implementation for n bit lengths

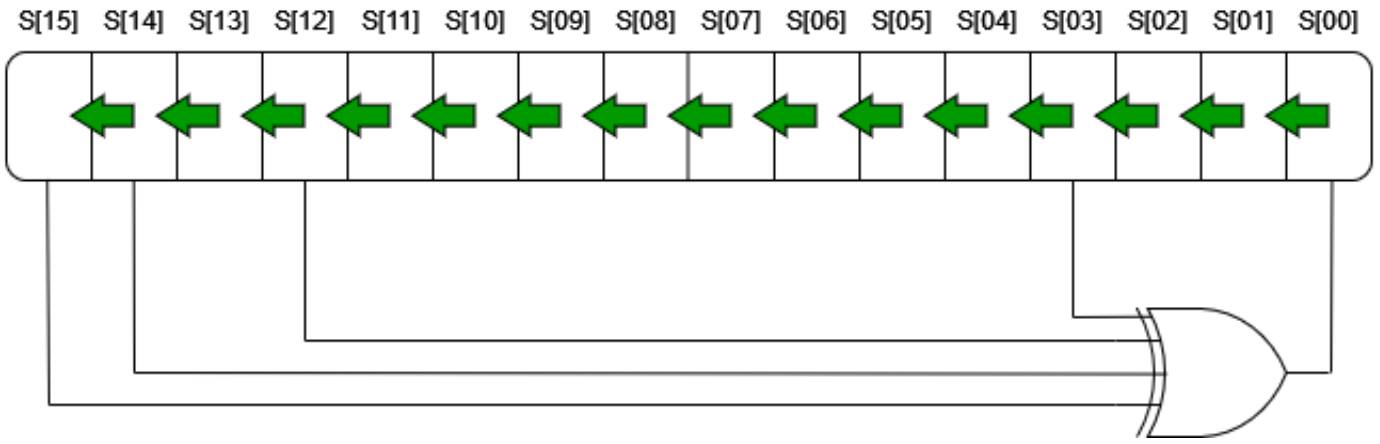

Figure 5: A 16 bit maximal LFSR.

4. Tasks

Most of the main modules have been given to you to help you understand the problem better. In fact, we have given much of the project for you in this text. The difference here is that we have not given you an testbenches and you will have to figure out how things go together. You have all the skills you need to complete this project, so trusting yourself and the process is worthwhile and I know all of you can do this. The tasks of the project are as follows:

1. Complete block diagrams of your system, include detailed interface specifications listing all signals and describing their timing.
2. Design a control logic presumably with a FSM to have it work correctly.
3. Build the testbench to simulate your design and make sure things are working for both the datapath, control and combined datapath/control.
4. Once your design completely works, implement the design on the DSDB board with the provided HDMI design. You should probably use the switches, push buttons, LEDs and also use the seven segment display to output key elements of your project. Remember, to use the LEDs to help you debug your design.

Again, the process here is not difficult. If you need to work out any of the procedures or ask me to inspect your design, I would recommend stopping by to ask questions or advice. I would not advise waiting until the last week to start as I might be busy with end-of-the-semester duties and starting early is the best practice.

4.1 Extra Credit

There are lots of opportunities for extra credit with this project. But, please, first focus on completing the baseline project before attempting the extra credit option. One of the advantages of digital logic is that many bits can be computed in parallel and then chosen later to be correct or incorrect.

Because Field Programmable Gate Arrays (FPGAs) can contain many millions of logic gates, you could easily expand this game beyond a 8×8 matrix computation. You will have to think about how to distribute this to greater dimensions, but its quite easy if you look at the files that are provided to you.

5. Video and Lab Report

I am asking for both a final report and video demo of your design. You can easily create a video on your cell phone that is no more than 5-10 minutes that encapsulates your design and how it works. Please work consistently throughout the final weeks of the semester to make sure you complete the project on time.

I will also give extra credit to those that put a little effort into making an outstanding video and showcasing their project in detail. You could also potentially discuss other topics including significant additions to your project.

You are also required to submit a final report of your design using the lab rubric. You should remember to submit both your lab report and video report to Canvas for your team, but please also submit your team evaluation, as well. Beware; no late projects will be accepted and if you miss submitting your project on time, you will receive a 0 for your project grade! This procedure should be similar to what you are using for your labs. You should also take a printout of your waveform from your ModelSim simulation. Only one of your team members should upload the files, lab report, and team assessment. Also, please make sure you hand in all files, including your HDL, testbenches, and other important files you wish for us to see.

Please contact Nate Lannan (nate.lannan@okstate.edu) for more help. Your code should be readable and well-documented. In addition, please turn in additional test cases or any other added item that you used. Please also remember to document everything in your Lab Report using the information found in the Grading Rubric.

References

[1] Martin Gardner, "The fantastic combinations of john conway's new solitaire game "life"," *Scientific American*, vol. 223, pp. 120–123, 1970.

[2] "Cleve's corner: Revisiting the game of life," `https://blogs.mathworks.com/cleve/2018/11/08/revisiting-the-game-of-life/#f8ed4bb9-8421-4ffd-90f0-f2e1b3f7fd29`, Accessed: 2022-11-03.

[3] Sarah Harris and David Harris, *Digital Design and Computer Architecture: RISC-V Edition*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2021.