Introduction to Digital Circuit Design
Term Project – Game of Life with FPGA
Tuna Alikaşifoğlu
EEE102 – 2
December 29, 2018

YouTube Link:

https://www.youtube.com/channel/UCMNvhhlCC_ulW6N2ptXAZuw/videos?view_as=subscriber

Abstract

In this project, implementation of John Conway's *Game of Life* is created on a FPGA board with VHDL. Despite the name, *Game of Life* is more like a simulation; user defines an initial condition which describes a cell population that will be simulated. No further input is needed to observe the consequences of the simulation. User defines an initial condition and observes how the system evolves. The rules of the simulation are determined by the Cambridge mathematician John Conway and these rules create a logical system for determining the fate of the cell population. Implementation of this logical system is handled by the FPGA board and the condition of the system is displayed on a monitor via VGA port of the board.

Design Specification Plan

Mechanics of the Game

In theory, the system of the Game of Life[1] is an infinite, two-dimensional orthogonal grid of cells. Although, as most of the implementations of the game, this one also can only show a segment of this infinite grid. Each grid represents a cell which can be

[1] See for more information: Martin Gardner, "Mathematical Games - the Fantastic Combinations of John Conway's New Solitaire Game "Life"," *Scientific American* 223 (1970).

in two possible states; either dead or alive. An example of a small population can provide a better understanding:

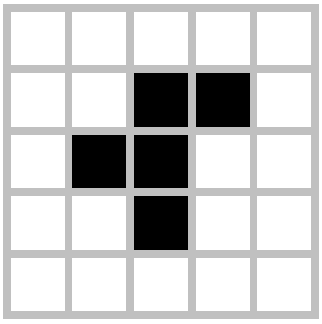

Figure 1: An example of cell population, The R-pentomino

Position of cells depends on their current state, user defines a position for the first generation of cells and next generations are created depending on the first one. Generations are created based on the position of the cells; each cell's fate depends on their neighbors. Every cell interacts with its eight neighbors which are the cells surround it. There are four basic rules to create the next generation of the cells which forms a logical system to implement. In his own implementation Edwin Martin summarizes these rules as the following:

- Each cell with one or no neighbors dies, as if by solitude.
- Each cell with four or more neighbors dies, as if by overpopulation.
- Each cell with two or three neighbors survives.
- Each cell with three neighbors becomes populated.[2]

[2] "John Conway's Game of Life," accessed December 29, 2018, https://bitstorm.org/gameoflife/.

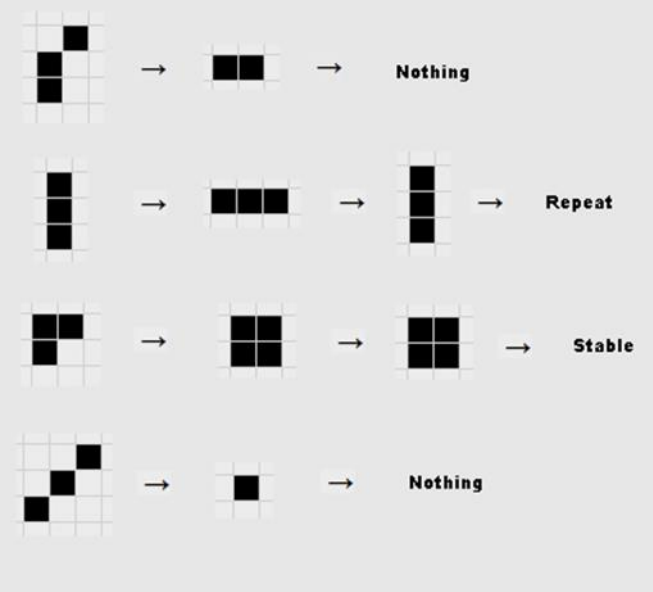

Figure 2: Possible scenarios for little populations

Idea Behind the Implementation

Since the environment of the system is a two-dimensional grid, using a two-dimensional array to implement the environment is the most straightforward solution. Moreover, since there are only two possible states for a cell, constraining the state of the cells by using "BIT" representation would be suitable. In this case whole system can be represented as two array of bits.

On the other hand, with every time-step this array must be updated. An algorithm is needed to change the bits according to the rules of the game. The four rules of the game can be simplified to create a simpler algorithm. Rules can be grouped to inspect the cells which are dead and alive. For alive cells, it can be said that they stay alive if they have two or three neighbors, otherwise they die. For the dead cells, they stay dead unless they have exactly three cells. With these simplified rules determining the next state of the

environment would be much easier. The next state of a single cell depends on the eight cells surrounds it, so determining a cell's number of neighbor is sufficient to calculate the next state that cell. Applying this method to all cells in the environment can determine the next state of the whole environment. With this determination the array of bits would be updated accordingly.

Finally, this array of bits is needed to be displayed as the user would like to see the output of the system. Since the implementation of the game is being created on a FPGA board, displaying the system via VGA port is a suitable solution. Overall, an environment would be created by two-dimensional array of bits to represent the universe of cells and this system would be displayed via VGA port of the FPGA port.

Design Methodology

Implementation of the Algorithm

As it stated in the previous section, fate of a cell depends only on the number neighbors of that specific cell. It is easy to implement algorithm for the whole environment if it can be implemented for just one cell. In this implementation this task is handled by using "nested for-loops". These technique is helpful to iterate over the two-dimensional array for the selected part, in this case eight cells around the specified cell. A visual example can be helpful, for the red cell selected, blue cells are checked:

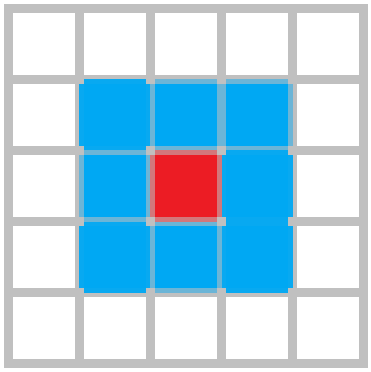

Figure 3: An example of the algorithm

However, since a two-dimensional array has been iterated, there are a few special cases to handle. Even though the theoretical game has an infinite grid as an environment, FPGA boards has finite locations to store bits. Thus, borders and the corners of the environment has less neighbors. For corners, three cells are checked; for borders five cells are checked where for any other cell eight cells are checked which are surrounding them.

After finding the number of neighbors for a cell, it is easy to find the next state of that cell. The simplified rules are applied using "if statements" to determine the fate of the cell. The whole algorithm is used inside another nested for-loop to determine the next state of every cell. Finally, calculated "next-state" is assigned to the current state, and everything starts over to calculate the new next-state.

Implementation of the Modular Design

For large projects like this one, it is more convenient to use modular design to handle the task more efficiently. There are lots of processes going on and it is hard to keep track all of them without a modular design. Large problems can be divided into smaller ones, and these small problems can be solved using modules. With the modular design,

these small problem solver modules can be connected. For example, in this project there are three main problems; first one is to take an initial position from the user, second is to update the system according to this condition and the third one is to display these changes via VGA port. These problems are handled in different modules and these modules are combined to create this project. In order to build the system as efficient as possible modular design technique is used for this project.

First of all, to be able to handle the project, the two-dimensional array, which represents the environment, must be passed between modules because of the fact that updating and displaying the array are handled with different modules. However, there is an issue with the declared array. To specify outputs of the multidimensional array, size of the array must be fixed and code must be hand-written in the form of one-dimensional arrays. Since it is more convenient for user to decide the size of the environment of the simulation it is not a desired design. Due to the limitations of the FPGA board, size of the environment cannot be dynamic; since it is an array, dynamic size arrays are not applicable in the VHDL. Despite the fact, it is preferred for this design to provide a changeable size for the user without having to write additional code for the desired system. With this design, size of the environment can be changed by only changing the value of a constant in the source code of top module of the project. To be able to do that, a new data type is defined for the two-dimensional array since in VHDL two-dimensional arrays cannot be specified as “out” in a module. By this technique number of cells can be changed by changing only an integer in the source code.

After solving the array problem, project is ready to implement the modular design. In this project, a package is declared in order to contain the declared data type as well as the functions for maintaining the environment. The functions of this package handles the process of updating the environment whenever they are called in the modules of the system. Besides the package that declared, there are four main modules under the top module that connects everything: “clock_div”, “environment”, “button_control” and “vga_controller”.

Implementation of the Clock Divider

The environment gets updated for every time-step, thus a clock divider is needed to specify this time-step for simulation. Moreover, different clock dividers are needed to implement push buttons and VGA display, these are discussed in the following sections. Due to the fact that clocks with different frequencies are needed, a general clock divider is created with generic input, so different modules can use desired clocks with the same implementation. For this implementation a counter is used which changes pre-built clock on the board to create slower clocks. Moreover, clock for the simulation is a dynamic clock, by creating a dynamic clock project offers an unfixed time-step for the simulation and user can specify time-step (four options is created for this project) by the aid of switches. For this project BASYS 3 is used as a FPGA board, which has a 100 MHz clock.

Implementation of the Environment

Environment module is the part that updates the system according to the time-step determined by the clock divider and also according to the inputs of the user. User can play and pause the simulation with a single switch. In order to start the simulation, an initial

condition must be set. System can be initialized by two different methods. First method by using switches user can select between a few pre-determined conditions (clear, all-alive, stable-four, glider, exploder, tumblr) set as default. Second method requires giving input by using push buttons (details are discussed in the next section). When the simulation is paused a cursor appears on the screen which can be used to iterate over the system. User can use this cursor with the help of the push buttons to add or remove a cell from the population. Since the push buttons on the BASYS 3 board forms an arrow shape, four outer buttons are used as up-left-right-down where the center button is used as the on-off switch. While paused user can iterate over the system and change the state of a cell which changes the specified bit in the array. User can change the state of multiple cells while paused. After the change, when the simulation is started again, the new system is perceived as a new initial condition and simulation continues from that point.

Implementation of the Push Buttons

For this project, taking input from the user to change the desired cells' state, using push buttons is the optimal way on a FPGA board. Push buttons are used to iterate over the environment and to change the state of desired cells. By doing so, users can create whatever initial condition they desire. However, implementation of push buttons is tricky on a FPGA board. Since every action depends on the clock of the system, timing problem is one of the most important problem. Besides, push buttons tend to bounce when they are clicked, which is another problem. To overcome these issues a button controller module is created. This module debounces the push buttons to make them work as they intended. When a

button is pressed, its value changes from ‘0’ to ‘1’. However, glitch may occur that causes changing between ‘0’ and ‘1’ even during this press. Usually, debouncers are implemented in a way that to prevent this glitch and to provide a steady state push. The debouncer that implemented in this project is a little bit different than the conventional debouncer. If a steady state push would have been used, the state of the cell would change rapidly during the press. Although, this is a way of implementing, it is not a preferred type of usage. In order to prevent this problem, a new debouncer is created to create a pulse when a button is pressed rather than a steady state signal. In order to that, three D flip-flops are used to create delays while their clock input is handled by using the clock divider to create button clock. Flip-flops suitable outputs are used to create a pulse. With this technique, it does not make a difference even if you click the button or push it for an extended period of time. With this implementation user can reverse the state of cell with one click.

Implementation of the VGA Controller

In this project output of the system is displayed via VGA port. VGA display have its own difficulties to overcome. First of all, a whole image cannot be displayed at once, image has to be traced one pixel at a time. With a high frequency clock that can trace every pixel, image can be seen as a whole to the human. For this project, 640x480 resolution is sufficient to display the output. For this purpose, a 25 MHz clock is needed to trace every pixel to generate 640x480, 60 Hz display, which handled by the clock divider. This module contains two sub-modules to display the environment on a monitor. First one is the VGA Regulator, this module handles the necessary implementations for the VGA display. First

of all, it contains two counters to iterate over the display both horizontally and vertically. Second, for 640x480 display, it handles front porch, back porch and sync pulse (see figure below). This module determines whether the display in the display zone or not. Finally, it passes each pixel at a time to the image source module called VGA display. Then, VGA display module decides the color of that specific pixel according to the two-dimensional array that has been passed by the environment module. It displays a grid on the monitor which has grey lines to separate each cell from one another. For example, if size of the environment is selected as twenty. User would face a 20x20 grid filled by rectangles. Size of these rectangles are designed to fill the entire screen, so for a 640x480 display, each cell is represented as 32x20 rectangles. A cell is painted to black if the cell in that location is dead and is painted to white if the cell is alive. Furthermore, it is said that user can iterate over the environment by push buttons. To able to do so user has to see the position of the cursor. When the simulation is paused, a cursor appears on the top left cell of the screen which is yellow. As user changes the position of the cursor, display changes the position of the cursor.

Results

As the following figures, suggests the algorithm works as the way intended, system works flawlessly for the pre-determined cases.

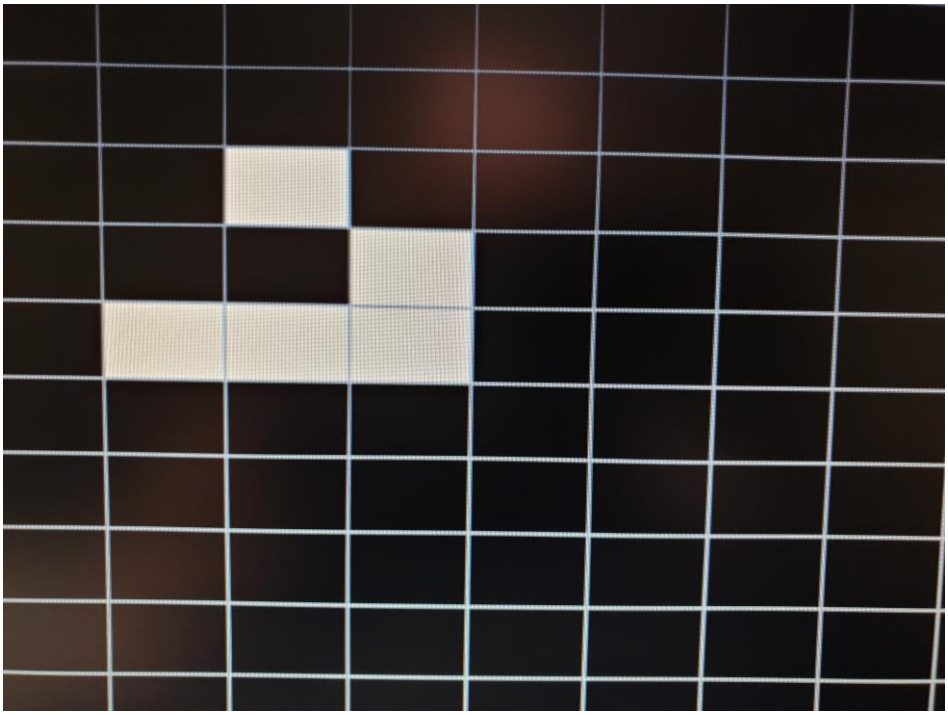

Figure 4: Glider, position 1

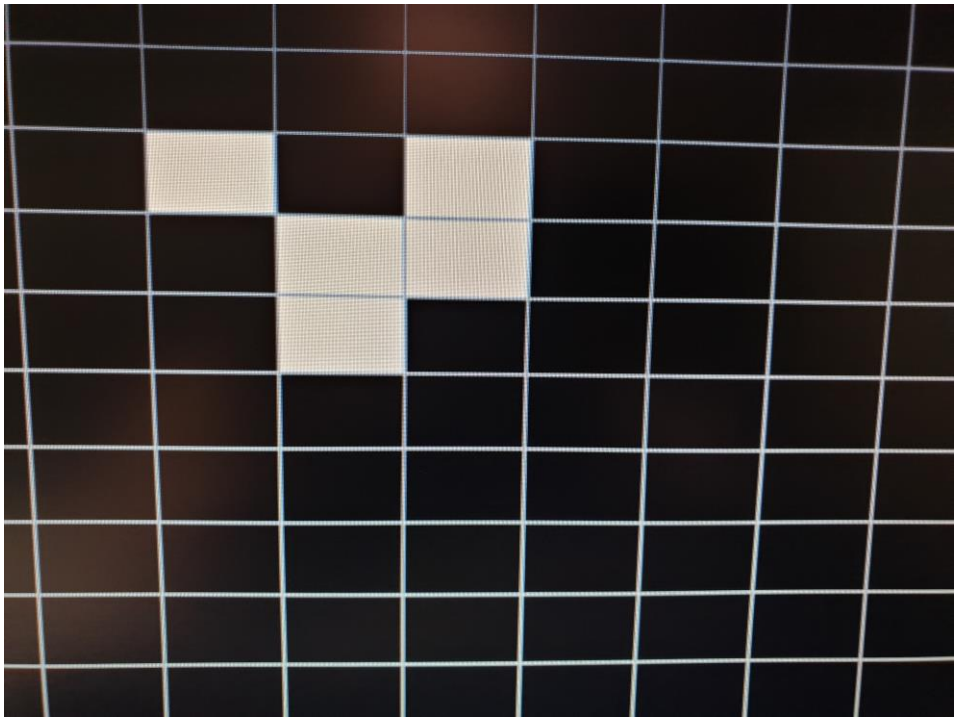

Figure 5: Glider, position 2

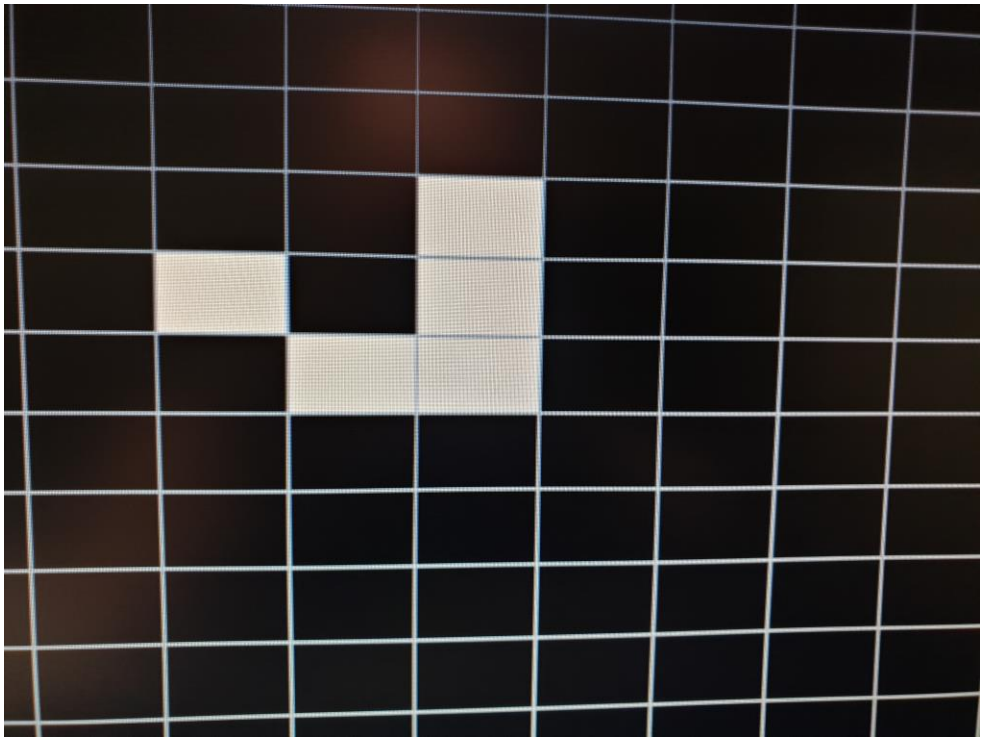

Figure 6: Glider, position 3

Moreover, as the following figures, suggests using the cursor in order to reverse the state of the cell is also functional.

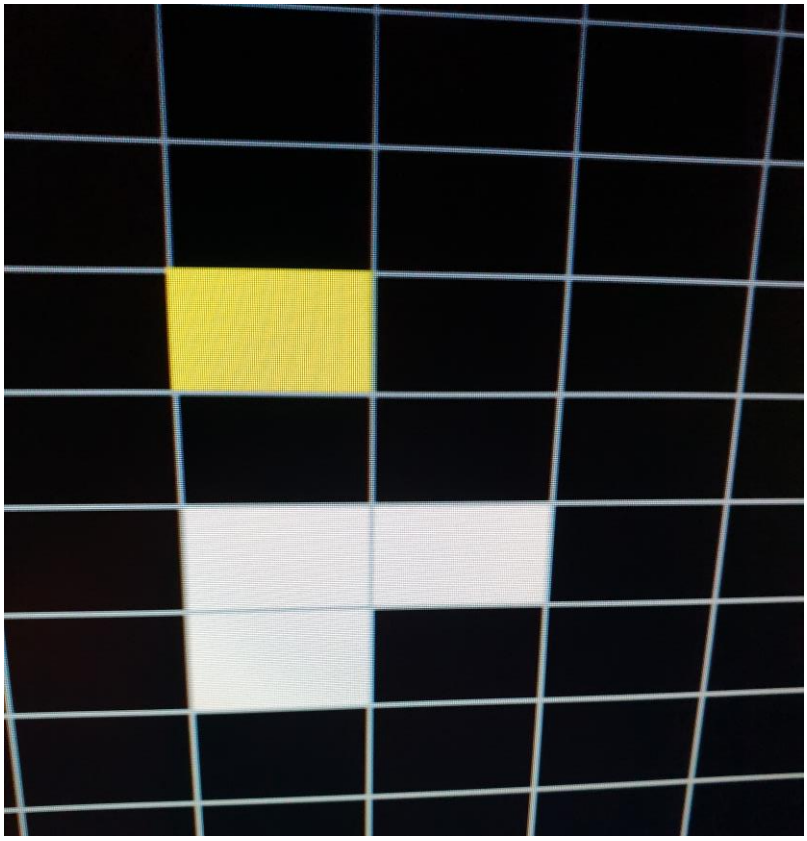

Figure 7: Reversing a cell, position 1

Figure 8: Reversing a cell, position 2

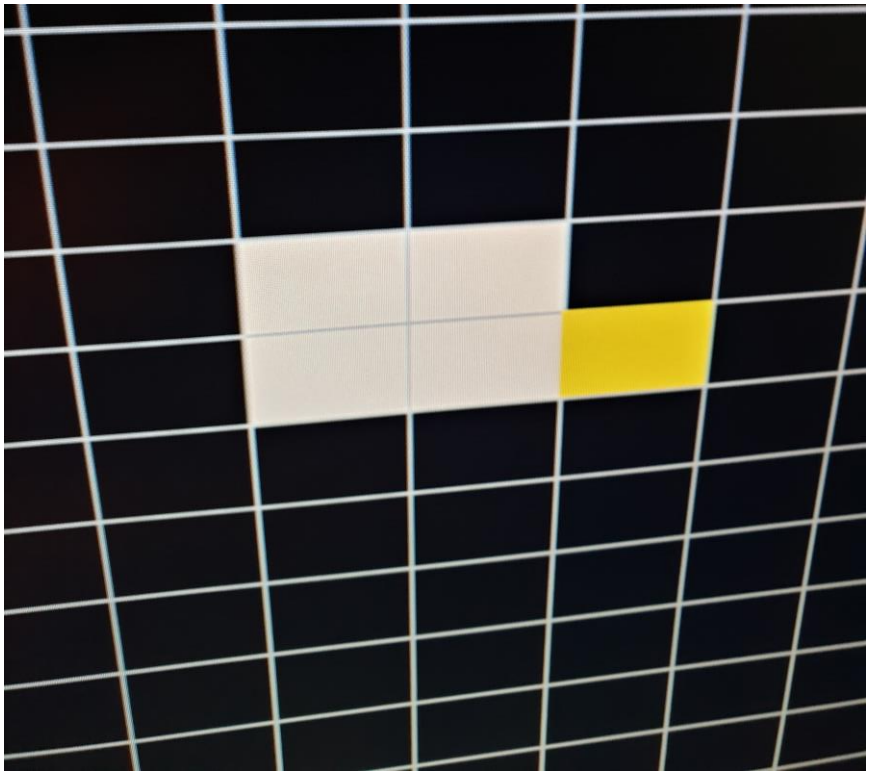

Figure 9: Reversing a cell, position 3

Conclusion

This project is my first relatively large scale VHDL project on a FPGA board. I designed this project as a term project for *EEE102: Introduction to Digital Design* course. I learned so much about digital design in this course yet I have much more to learn. In this course, aim of this project was to get familiar with VHDL via FPGA boards, in order to implement our theoretical knowledge from courses into something that we want design. *Game of Life* is one the common coding challenges in software programming. I believe, I managed to implement it very well in a hardware design. Process of creating this project is distributed on the semester, so there are probably parts that I design differently with my current knowledge. For example, instead of using two-dimensional array, a RAM implementation can be more efficient for this design. In order to

implement the whole project, I needed to use the concepts covered in the course. From combinational circuits to finite state machines, D flip-flops to counters, everything that I learned in the class helped me to achieve Game of Life on a FPGA board. I'm glad that I am at state that I can design functional projects with VHDL at the end of the semester.

Bibliography

Gardner, Martin. "Mathematical Games - the Fantastic Combinations of John Conway's New Solitaire Game "Life"." *Scientific American* 223 (1970): 120-23.

"John Conway's Game of Life." accessed December 29, 2018, https://bitstorm.org/gameoflife/.

Appendices

Appendix A: Data Sheet of BASYS 3

https://reference.digilentinc.com/basys3/refmanual

Appendix B: VHDL Source Code

Top Module:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use WORK.ENVIRONMENT_PACKAGE.ALL;


entity top_module is
    Port (  clk100mhz          : in STD_LOGIC;
            initial_condition   : in STD_LOGIC_VECTOR(2 downto 0);
            enable              : in STD_LOGIC;
            time_selection      : in STD_LOGIC_VECTOR(1 downto 0);
            size_selection      : in STD_LOGIC_VECTOR(1 downto 0);
            button              : in STD_LOGIC_VECTOR(4 downto 0);
```

```
        leds            : out STD_LOGIC_VECTOR(7 downto 0);
        h_sync     : out STD_LOGIC;
        v_sync      : out STD_LOGIC;
        vgaRed          : out STD_LOGIC_VECTOR(3 downto 0);
        vgaGreen        : out STD_LOGIC_VECTOR(3 downto 0);
        vgaBlue         : out STD_LOGIC_VECTOR(3 downto 0));
end top_module;

architecture Behavioral of top_module is
   -- components
   component environment is
      Generic (FIELD_SIZE : INTEGER := 4);
      Port    (clock          : in STD_LOGIC;
               button_clock    : in STD_LOGIC;
               clock_enable    : in STD_LOGIC;
               button          : in STD_LOGIC_VECTOR(4 downto 0);
```

```
        initial_condition : in STD_LOGIC_VECTOR(2 downto 0);



        cursor_x          : out INTEGER;

        cursor_y          : out INTEGER;



        field_out         : out FIELD(0 to (FIELD_SIZE - 1), 0 to (FIELD_SIZE - 1)));
end component;



component vga_controller is
   Generic (FIELD_SIZE : INTEGER := 4);
   Port (  clock100mhz     : in STD_LOGIC;
           clock_enable    : in STD_LOGIC;
           cursor_x        : in INTEGER;
           cursor_y        : in INTEGER;
           myField         : in FIELD(0 to FIELD_SIZE - 1, 0 to FIELD_SIZE - 1);
           size_selection  : in STD_LOGIC_VECTOR(1 downto 0);
```

```
        h_sync : out STD_LOGIC;
        v_sync   : out STD_LOGIC;
        red            : out STD_LOGIC_VECTOR(3 downto 0);
        green          : out STD_LOGIC_VECTOR(3 downto 0);
        blue           : out STD_LOGIC_VECTOR(3 downto 0));
end component;

component clock_div is
   Generic (COUNTER_BIT : INTEGER);
   Port    (selection    : in INTEGER;
            clock_in     : in STD_LOGIC;
            reset        : in STD_LOGIC;
            clock_enable : in STD_LOGIC := '1';
            clock_out    : out STD_LOGIC);
end component;

component button_control is
```

```
    Port (  clock100mhz  : in STD_LOGIC;
            button       : in STD_LOGIC_VECTOR(4 downto 0);
            selection    : in INTEGER;
            clock_button : out STD_LOGIC;
            button_out   : out STD_LOGIC_VECTOR(4 downto 0));
end component;
-- constants
constant SIZE         : INTEGER := 20;
constant clock_bit    : INTEGER := 27;


-- signals
signal clock_reset    : STD_LOGIC := '0';
signal clock_out      : STD_LOGIC;
signal myField        : FIELD(0 to (SIZE - 1), 0 to (SIZE - 1));
signal clock_division : INTEGER;


signal button_clock   : STD_LOGIC;
```

```
    signal button_out     : STD_LOGIC_VECTOR(4 downto 0);



    signal cursor_control : STD_LOGIC_VECTOR(4 downto 0);

    signal cursor_x       :INTEGER;

    signal cursor_y       :INTEGER;


begin

    -- identify which switches are on----------

    leds(0) <= enable;

    leds(3 downto 1) <= initial_condition;

    leds(5 downto 4) <= size_selection;

    leds(7 downto 6) <= time_selection;

    ---------------------------------------------



    -- extra aid for managing modular design

    clock_division <= 2 * to_integer(unsigned(time_selection));

    cursor_control  <= button_out(4 downto 1) & button(0);
```

```
P0 : clock_div
   Generic Map ( COUNTER_BIT  => 27 )
   Port Map    ( selection    => clock_division,
                 clock_in     => clk100mhz,
                 reset        => clock_reset,
                 clock_out    => clock_out);

P1 : button_control
   Port Map    (clock100mhz   => clk100mhz,
                button        => button,
                selection     => clock_division,
                clock_button  => button_clock,
                button_out    => button_out);

P2 : environment
Generic Map (FIELD_SIZE => SIZE)
```

```
Port Map    (clock            => clock_out,
             button_clock      => button_clock,
             clock_enable      => enable,
             button            => cursor_control,
             initial_condition => initial_condition,
             cursor_x          => cursor_x,
             cursor_y          => cursor_y,
             field_out         => myField);


P3 : vga_controller
   Generic Map (FIELD_SIZE => SIZE)
   Port Map (  clock100mhz     => clk100mhz,
               clock_enable    => enable,
               cursor_x        => cursor_x,
               cursor_y        => cursor_y,
               size_selection  => size_selection,
               myField         => myField,
```

```
            h_sync => h_sync,
            v_sync   => v_sync,
            red           => vgaRed,
            green          => vgaGreen,
            blue           => vgaBlue);
end Behavioral;
```

Clock Divider:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


entity clock_div is
   Generic (COUNTER_BIT : INTEGER);
   Port    (selection    : in INTEGER := 0;
            clock_in     : in STD_LOGIC;
            reset        : in STD_LOGIC;
            clock_enable : in STD_LOGIC := '1';
```

```
        clock_out    : out STD_LOGIC);

end clock_div;



architecture Behavioral of clock_div is

   signal counter : UNSIGNED((COUNTER_BIT - 1) downto 0) := to_unsigned(0 , COUNTER_BIT);

begin

   process( clock_in , reset )

   begin

      if (reset = '1') then

         counter <= to_unsigned(0 , COUNTER_BIT);

      elsif rising_edge(clock_in) then

         if clock_enable = '1' then

            counter <= counter + 1;

         end if;

      end if;

   end process;
```

```
    clock_out <= STD_LOGIC( counter(COUNTER_BIT - selection) );

end Behavioral;
```

Button Control:

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;



entity button_control is

    Port (  clock100mhz  : in STD_LOGIC;

            button       : in STD_LOGIC_VECTOR(4 downto 0);

            selection    : in INTEGER;

            clock_button : out STD_LOGIC;

            button_out   : out STD_LOGIC_VECTOR(4 downto 0));

end button_control;



architecture Behavioral of button_control is

-- components
```

```
    component button_regulator is
        Port (  clock : in STD_LOGIC;
                btn   : in STD_LOGIC_VECTOR(4 downto 0);
                push  : out STD_LOGIC_VECTOR(4 downto 0));
    end component;

    component clock_div is
        Generic (COUNTER_BIT : INTEGER);
        Port    (selection    : in INTEGER := 0;
                 clock_in     : in STD_LOGIC;
                 reset        : in STD_LOGIC;
                 clock_enable : in STD_LOGIC;
                 clock_out    : out STD_LOGIC);
    end component;

-- signals
    signal enable : STD_LOGIC := '1';
```

```
    signal reset  : STD_LOGIC := '0';



    signal button_clock : STD_LOGIC;

    signal button_click : STD_LOGIC_VECTOR(4 downto 0);



begin

    clock_button <= button_clock;



    P1 : clock_div

       Generic Map (COUNTER_BIT  => 26)

       Port Map    (selection    => selection,

                    clock_in     => clock100mhz,

                    reset        => reset,

                    clock_enable => enable,

                    clock_out    => button_clock);
```

```
   P2 : button_regulator

      Port Map (  clock  => button_clock,

                  btn    => button,

                  push   => button_out);



end Behavioral;
```

Button Regulator:

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;



entity button_regulator is

   Port (  clock : in STD_LOGIC;

           btn   : in STD_LOGIC_VECTOR(4 downto 0);

           push  : out STD_LOGIC_VECTOR(4 downto 0));

end button_regulator;



architecture Behavioral of button_regulator is
```

```
    signal delay1, delay2, delay3 : STD_LOGIC_VECTOR(4 downto 0);

begin



    push_pulse : process(clock)

    begin

        if rising_edge(clock) then

            delay1 <= btn;

            delay2 <= delay1;

            delay3 <= delay2;

        end if;

    end process;



    push <= delay1 AND delay2 AND (not delay3);



end Behavioral;
```

Environment:

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;

use WORK.ENVIRONMENT_PACKAGE.ALL;



entity environment is

   Generic (FIELD_SIZE : INTEGER := 4);

   Port    (clock            : in STD_LOGIC;

            button_clock     : in STD_LOGIC;

            clock_enable     : in STD_LOGIC;

            button           : in STD_LOGIC_VECTOR(4 downto 0);

            initial_condition : in STD_LOGIC_VECTOR(2 downto 0);



            cursor_x         : out INTEGER;

            cursor_y         : out INTEGER;

            field_out        : out FIELD(0 to (FIELD_SIZE - 1), 0 to (FIELD_SIZE - 1)));

end environment;
```

```
architecture Behavioral of environment is

    -- signals

    -- next state

    signal myField   : FIELD(0 to (FIELD_SIZE - 1), 0 to (FIELD_SIZE - 1));

    signal newField  : FIELD(0 to (FIELD_SIZE - 1), 0 to (FIELD_SIZE - 1));

    signal condition : STD_LOGIC_VECTOR(2 downto 0) := "000";

    signal selection : STD_LOGIC_VECTOR(1 downto 0);

    -- cursor

    signal change              : STD_LOGIC := '0';

    signal change_order        : STD_LOGIC := '0';

    signal column              : INTEGER range 0 to FIELD_SIZE - 1 := 0;

    signal row                 : INTEGER range 0 to FIELD_SIZE - 1 := 0;

    signal delay1, delay2, delay3 : STD_LOGIC;

--------------------------------------------------------------------------

begin

    selection <= (clock_enable & change_order);

    with selection select newField <= update(myField) when "10",
```

```
                update(myField) when "11",
                reverse(myField, column, row) when "01",
                myField         when others;


field_out <= myField;


change_order <= delay1 AND delay2 AND (not delay3);
cursor_x <= column;
cursor_y <= row;


NEXT_STATE: process(clock, initial_condition)
begin
   if rising_edge(clock) then

      if(initial_condition = condition) then
         myField <= newField;
      else
```

```
            myField <= initialize(myField, initial_condition);

            condition <= initial_condition;

        end if;

    end if;

end process;



    CURSOR : process(button_clock)

begin

    if rising_edge(button_clock) then

        change <= '0';

        if clock_enable = '1' then

            column <= 0;

            row    <= 0;



        -- down button

        elsif button = "10000" then

            if (row = FIELD_SIZE - 1) then
```

```
            row <= 0;
        else
            row <= row + 1;
        end if;
    -- right button
    elsif button = "01000" then
        if (column = FIELD_SIZE - 1) then
            column <= 0;
        else
            column <= column + 1;
        end if;
    -- left button
    elsif button = "00100" then
        if (column = 0) then
            column <= FIELD_SIZE - 1;
        else
            column <= column - 1;
```

```
            end if;

        -- up button

        elsif button = "00010" then

            if (row = 0) then

                row <= FIELD_SIZE - 1;

            else

                row <= row - 1;

            end if;



        -- center button

        elsif button = "00001" then

            change <= '1';

        end if;

    end if;

end process;



push_pulse : process(clock)
```

```
    begin

        if rising_edge(clock) then

            delay1 <= change;

            delay2 <= delay1;

            delay3 <= delay2;

        end if;

    end process;



end Behavioral;
```

VGA Controller:

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use WORK.ENVIRONMENT_PACKAGE.ALL;



entity vga_controller is

    Generic (FIELD_SIZE : INTEGER := 4);

    Port (  clock100mhz     : in STD_LOGIC;
```

```
        clock_enable    : in STD_LOGIC;
        cursor_x        : in INTEGER;
        cursor_y        : in INTEGER;
        myField         : in FIELD(0 to FIELD_SIZE - 1, 0 to FIELD_SIZE - 1);
        size_selection  : in STD_LOGIC_VECTOR(1 downto 0);

        h_sync : out STD_LOGIC;
        v_sync   : out STD_LOGIC;
        red            : out STD_LOGIC_VECTOR(3 downto 0);
        green          : out STD_LOGIC_VECTOR(3 downto 0);
        blue           : out STD_LOGIC_VECTOR(3 downto 0));
end vga_controller;

architecture Behavioral of vga_controller is
-- components ----------------------------
component clock_div
   Generic (COUNTER_BIT : INTEGER);
```

```
    Port    (selection    : in INTEGER := 0;
             clock_in     : in STD_LOGIC;
             reset        : in STD_LOGIC;
             clock_enable : in STD_LOGIC;
             clock_out    : out STD_LOGIC);
end component;

component vga_regulator
          Generic(
          H_PIXELS     : INTEGER   := 640;
          H_PULSE      : INTEGER   := 96;
          H_FRONTPORCH : INTEGER   := 16;
          H_BACKPORCH  : INTEGER   := 48;
          H_POLARITY   : STD_LOGIC := '0';

          V_PIXELS       : INTEGER   := 480;
          V_PULSE        : INTEGER   := 2;
```

```
        V_FRONTPORCH   : INTEGER   := 10;
        V_BACKPORCH    : INTEGER   := 33;
        V_POLARITY     : STD_LOGIC := '0');
    Port (  clock, clear          : in STD_LOGIC;
        h_sync_pulse : out STD_LOGIC;
        v_sync_pulse   : out STD_LOGIC;
        h_out        : out INTEGER;
        v_out         : out INTEGER;
        display_enable        : out STD_LOGIC);
end component;

component vga_display
    Generic (FIELD_SIZE : INTEGER := 4);
    Port (  display_enable      : in STD_LOGIC;
        clock_enable        : in STD_LOGIC;

        cursor_x            : in INTEGER;
```

```
        cursor_y          : in INTEGER;



        column            : in INTEGER;

        row               : in INTEGER;

        thickness_selection : in STD_LOGIC_VECTOR(1 downto 0);

        myField           : in FIELD(0 to FIELD_SIZE - 1, 0 to FIELD_SIZE - 1);



        red               : out STD_LOGIC_VECTOR(3 downto 0);

        green             : out STD_LOGIC_VECTOR(3 downto 0);

        blue              : out STD_LOGIC_VECTOR(3 downto 0));

end component;



-- signals --------------------------------

-- clock divider --

signal clock_reset : STD_LOGIC := '0';

signal clock25mhz : STD_LOGIC;

signal enable : STD_LOGIC := '1';
```

```
-- vga regulator --

signal h_outS : INTEGER;

signal v_outS : INTEGER;

signal dis_en : STD_LOGIC;



begin

   P1 : clock_div

   Generic Map ( COUNTER_BIT  => 2 )

   Port Map    ( selection    => 1,

                 clock_in     => clock100mhz,

                 reset        => clock_reset,

                 clock_enable => enable,

                 clock_out    => clock25mhz);



   P2: vga_regulator

   Port Map (  clock              => clock25mhz,

               clear              => clock_reset,
```

```
        h_sync_pulse => h_sync,
        v_sync_pulse   => v_sync,
        h_out       => h_outS,
        v_out        => v_outS,
        display_enable       => dis_en);


P3: vga_display
Generic Map (FIELD_SIZE => FIELD_SIZE)
Port Map (  display_enable     => dis_en,
        clock_enable       => clock_enable,
        cursor_x          => cursor_x,
        cursor_y          => cursor_y,
        column            => h_outS,
        row               => v_outS,
        thickness_selection => size_selection,
        myField           => myField,
        red               => red,
```

```
        green           => green,
        blue            => blue);
end Behavioral;
```

VGA Regulator:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity vga_regulator is
    Generic(
    H_PIXELS     : INTEGER   := 640;
    H_PULSE      : INTEGER   := 96;
    H_FRONTPORCH : INTEGER   := 16;
    H_BACKPORCH  : INTEGER   := 48;
    H_POLARITY   : STD_LOGIC := '0';

    V_PIXELS     : INTEGER   := 480;
    V_PULSE      : INTEGER   := 2;
```

```
        V_FRONTPORCH   : INTEGER   := 10;

        V_BACKPORCH    : INTEGER   := 33;

        V_POLARITY     : STD_LOGIC := '0');



    Port (  clock, clear          : in STD_LOGIC;

        h_sync_pulse : out STD_LOGIC;

        v_sync_pulse   : out STD_LOGIC;

        h_out        : out INTEGER;

        v_out         : out INTEGER;

        display_enable        : out STD_LOGIC);

end vga_regulator;



architecture Behavioral of vga_regulator is

    -- constants --

        constant H_PERIOD   : INTEGER := H_PIXELS + H_PULSE

                                    + H_FRONTPORCH + H_BACKPORCH;
```

```
        constant V_PERIOD     : INTEGER := V_PIXELS + V_PULSE

                                  + V_FRONTPORCH + V_BACKPORCH;

    -- signals --

    signal v_count_enable : STD_LOGIC := '0';

begin



    process( clock , clear )

        variable h_count : INTEGER range 0 to (H_PERIOD - 1) := 0;

        variable v_count   : INTEGER range 0 to (V_PERIOD - 1)   := 0;

    begin

        if (clear = '1') then

            h_count      := 0;

            v_count       := 0;

            display_enable       <= '0';

            h_sync_pulse <= not H_POLARITY;

            v_sync_pulse   <= not V_POLARITY;
```

```
elsif rising_edge( clock ) then

   -- horizontal pixel count

   if (h_count = (H_PERIOD - 1)) then

      h_count := 0;

      v_count_enable <= '1';

   else

      h_count := h_count + 1;

      v_count_enable <= '0';

   end if;



   -- vertical pixel count

   if (v_count_enable = '1') then

      if (v_count = (V_PERIOD - 1)) then

         v_count := 0;

      else

         v_count := v_count + 1;

      end if;
```

```
end if;



-- generate horizontal sync pulse

if ( h_count >= H_PERIOD - (H_PULSE + H_BACKPORCH) AND

     h_count < H_PERIOD - H_BACKPORCH ) then

    h_sync_pulse <= H_POLARITY;

else

    h_sync_pulse <= not H_POLARITY;

end if;



-- generate vertical sync pulse

if ( v_count >= V_PERIOD - (V_PULSE + V_BACKPORCH) AND

     v_count < V_PERIOD - V_BACKPORCH ) then

    v_sync_pulse <= V_POLARITY;

else

    v_sync_pulse <= not V_POLARITY;

end if;
```

```
        -- enable display
        if (h_count < H_PIXELS AND v_count < V_PIXELS ) then
            display_enable <= '1';
        else
            display_enable <= '0';
        end if;


        -- generate each pixel
        if (h_count < H_PIXELS) then
            h_out <= h_count;
        end if;
        if (v_count < V_PIXELS) then
            v_out <= v_count;
        end if;
    end if;
end process;
```

```
end Behavioral;
```

VGA Display:

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use WORK.ENVIRONMENT_PACKAGE.ALL;

--use IEEE.NUMERIC_STD.ALL;

--use UNISIM.VComponents.all;



entity vga_display is

   Generic (FIELD_SIZE : INTEGER := 4);

   Port (  display_enable      : in STD_LOGIC;

           clock_enable        : in STD_LOGIC;



           cursor_x            : in INTEGER;

           cursor_y            : in INTEGER;
```

```
        column              : in INTEGER;
        row                 : in INTEGER;
        thickness_selection : in STD_LOGIC_VECTOR(1 downto 0);
        myField             : in FIELD(0 to FIELD_SIZE - 1, 0 to FIELD_SIZE - 1);

        red                 : out STD_LOGIC_VECTOR(3 downto 0);
        green               : out STD_LOGIC_VECTOR(3 downto 0);
        blue                : out STD_LOGIC_VECTOR(3 downto 0));
end vga_display;

architecture Behavioral of vga_display is
    -- signals
    signal row_thickness_pixel : INTEGER;
    signal col_thickness_pixel : INTEGER;
    signal indexX : INTEGER;
    signal indexY : INTEGER;
```

```
begin


    with thickness_selection select

    row_thickness_pixel <=  24 when "00",

                            48 when "01",

                            60 when "10",

                            120 when "11";


    with thickness_selection select

    col_thickness_pixel <=  32 when "00",

                            64 when "01",

                            80 when "10",

                            160 when "11";


    indexX <= (column / col_thickness_pixel);

    indexY <= (row / row_thickness_pixel);
```

```
process(display_enable, column, row, thickness_selection)

begin

   if (display_enable = '1') then

      if (row    mod row_thickness_pixel = 0 OR

         column mod col_thickness_pixel = 0)  then

         -- grey

         red   <= "1000";

         green <= "1000";

         blue  <= "1000";

      elsif ((indexX < myField'length) AND (indexY < myField'length)

            AND isAlive(myField, indexX, indexY)) then

         -- white

         red   <= (others => '1');

         green <= (others => '1');

         blue  <= (others => '1');

      else

         -- black
```

```
        red   <= (others => '0');
        green <= (others => '0');
        blue  <= (others => '0');
    end if;


    if clock_enable = '0' then
        if (indexX = cursor_x AND indexY = cursor_y)  then
            -- yellow
            red   <= (others => '1');
            green <= (others => '1');
            blue  <= (others => '0');
        end if;
    end if;
else
    -- black
    red   <= (others => '0');
    green <= (others => '0');
```

```
            blue  <= (others => '0');

        end if;

    end process;

end Behavioral;
```