Tiny World: A Verilog Life Simulation Game

Albert Garcia, Mario Peraza, Nailah Smith

12/13/20
6.111 Introductory Digital Systems Laboratory
Group 24
Joe Steinmeyer

Abstract

Tiny World is a life simulation videogame implemented in System Verilog using an FPGA and its built-in switches and buttons. The game consists of dynamic components such as people, food, buildings, and water which move and react to user inputs. Different user inputs affect people in varied ways. Users can trigger floods which cause people and buildings to be removed from the game. Using switches, the user can vary the randomness of the game and food storage of food zones. The people continue to expand and grow based on the different situations encountered. All graphics are displayed on a monitor using VGA connection.

1 Overview

We implemented a life simulation, displayed on a monitor using VGA-HDMI. The initial set-up of the display consists of an island, water, 2 food zones, 24 buildings, and 16 people randomly placed on the island using a random number generator (figure 1). After set-up, people move through the island, colliding with other people to spawn another person randomly on the screen. People have individual attributes stored with them which change each round, such as age, movement direction, and food count. When a person's age surpasses a determined number, or when their food count is too low, the person is removed from the screen. By pressing a button, water floods the island, removing any buildings and/or people that are covered by the water.

Figure 1: Monitor showing the game display of the island (yellow) in the middle of the screen, water (blue) on the left and right sides of the island, 2 food zones (green) in the center of the island, 24 buildings (dark/light brown) at the top and bottom of the screen, and 16 people (red) scattered throughout the island.

2 Implementation

The concept of a grid was used to determine locations of dynamic objects in the project. A single grid space represents 8x8 pixels. A grid can be thought of as an array of a certain length where each index represents a grid space. The island is a 96x72 grid meaning it contains 6912 grid spaces. For the purposes of this project these indices were mapped to the addresses of two BRAMs of size 8192. The reason the grid is represented using BRAMs instead of a single array is to allow for the storage of information about objects such as people and food. Grid spaces may correspond to people which have attributes such as age and direction which need to be mapped and carried with them. The use of two BRAMs allows the game to read and write to separate BRAMs, essentially having a current grid being updated and an old grid with past information on it.

3 Modules and Block Diagram

Three main modules--top level, movement & collision, and drawing--handled most of the game logic, where drawing and movement & collision used several submodules to help break down their tasks. Top Level helps to designate the required inputs to each module where the other two main modules output information needed for a user to see the game (figure 2).

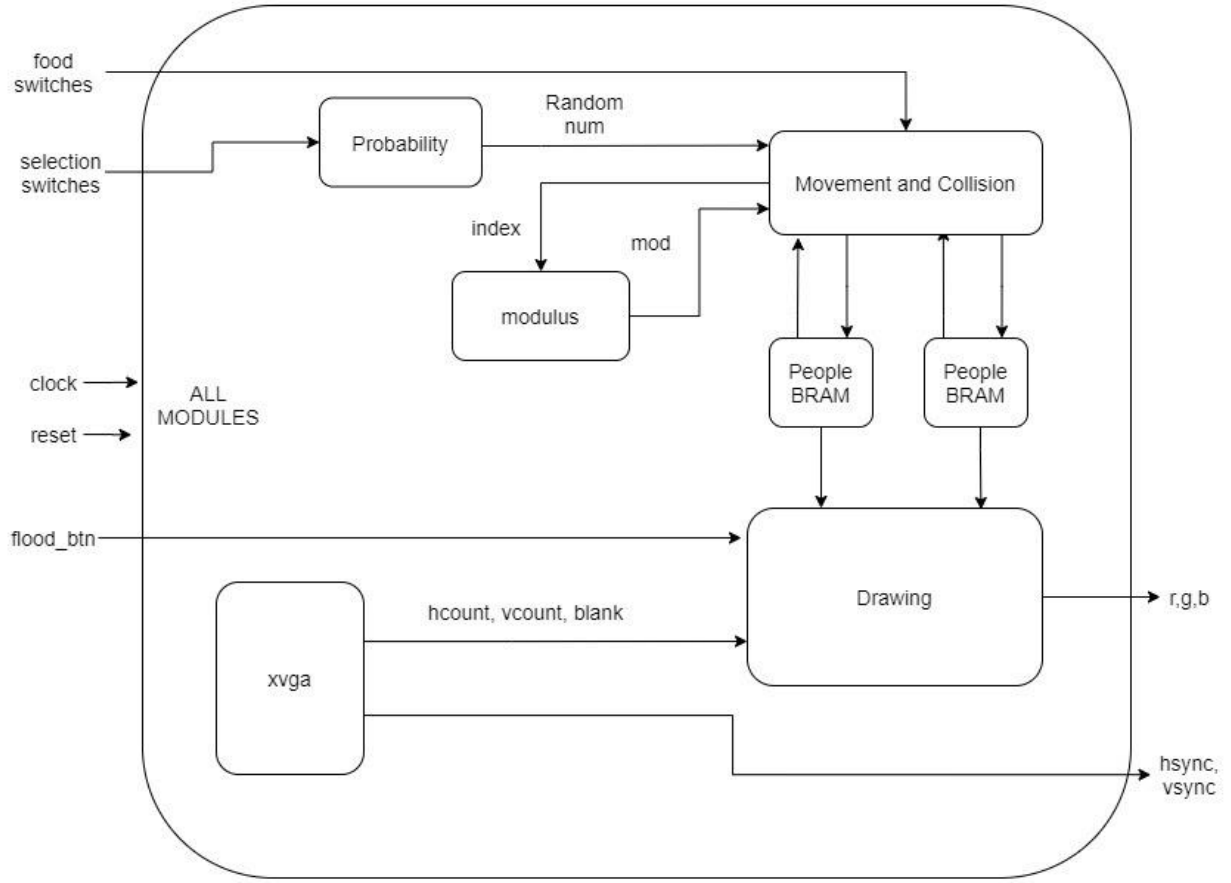

Figure 2: Diagram of all modules showing inputs coming from Top Level and outputs going to Top Level. Also shows how each module interacts with each other.

3.1 Top Level (Albert)

In the Top Level module are the BRAMs and instantiations of our high-level modules. Top Level was mostly used to send BRAM data back and forth between Movement & Collision and Drawing. We used two BRAMS which stored our grid information, with a size large enough to store our 96x72 grid. The Movement & Collision module wrote and read from the BRAMs. It read from the "old" BRAM, which was the BRAM altered in the previous game step, and used the information retrieved to write to the "new" BRAM (figure 3). The Drawing module only read from the BRAMs; however, to prevent any errors with writing to/reading from the same BRAM in the same clock cycle, we sent a signal from Movement & Collision to Top Level that told the Drawing module which BRAM to read from. We chose to keep the maximum size of 8192 for the BRAMs, intending to use the extra space as storage for people inside the buildings. Each spot in the BRAM had a depth of 30 bits, with 8 bits reserved for food information and 22 bits reserved for people. For the people information, 7 bits were used for storing food (food_count), 7 bits for age, 5 bits for the person's counter (which would have been used to keep track of how long until a person should exit a building), and 3 bits for direction. However, in the final implementation, the people had a maximum of four directions rather than eight, so direction was reduced from 3 bits to 2 bits.

Seven of the food bits were used for storing the current amount of food that the food zone held (food capacity), and 1 bit was used as an indicator (food_indicator) for the Drawing module to know whether a food zone should be drawn there. In our final implementation, we did not use these 8 bits for food, since instead of using many small 2x2 grid zones, we opted for 2 large food zones, whose information could just be stored in a 10-bit array (food_storage1 and 2).

Top Level also had a logic value game_step which was sent to both Movement & Collision and Drawing. We originally used game_step as a method for debugging, with game_step triggering at a button push. This allowed for examination of each stage of the simulation to see if people were moving as expected. In the final implementation, game_step was updated to be triggered after a certain number of cycles. This was convenient as well, since we could change the speed of the game as needed by changing the number of cycles to count.

Additionally, Top Level sent the hsync, vsync, and blank output values of the XVGA module to the Drawing module. A clock divider was also implemented in top level, used both for the timing at which graphics were drawn to the screen, and to run the game logic at a slower speed of 65mhz rather than 100mhz.

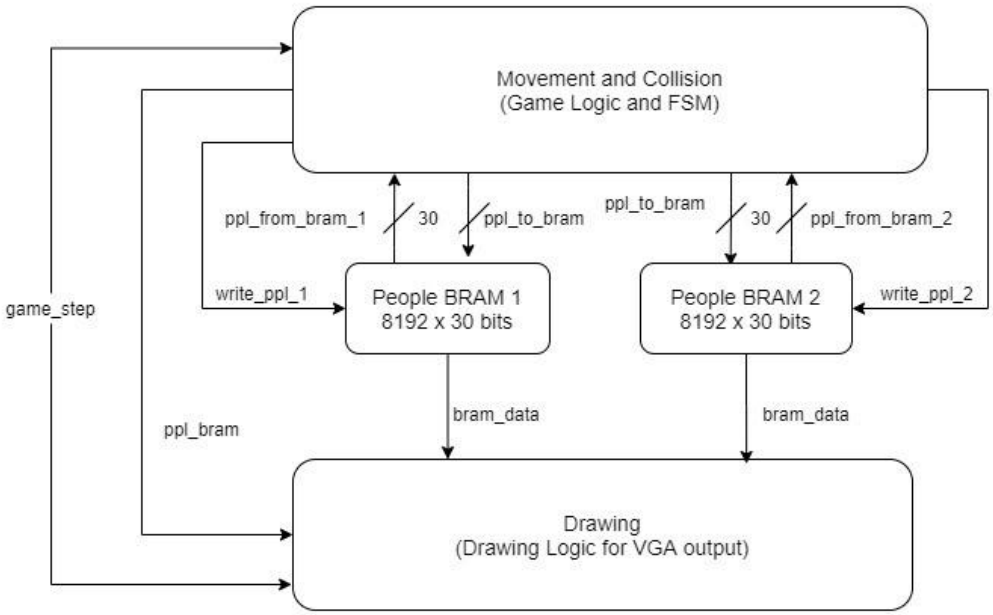

Figure 3: Block diagram showing the data which gets sent to and from the modules and BRAMs.

3.2 Movement and Collision (All)

Designing the Movement and Collision (M&C) module was a bit difficult because it managed all the game logic, such as moving people each game step, determining when to increment a person's food count, and checking for island bounds. If a person moved into a food zone their food count would increment, otherwise their food count would be constantly decrementing unless they stayed still. The M&C module communicated with the BRAMs in Top Level by sending information and reading information about people each game step. Since the Drawing module also needed to read from the BRAMs to draw people onto the screen we alternated which one we wrote to in the M&C module. This was done by sending a signal (ppl_bram) to the Drawing module so it knew which BRAM to read from. M&C kept track of two arrays, one that was used to determine what address to send (lookup_addr) -- dependent on the current index of the person being evaluated in the state machine -- and one that was used to store information retrieved from BRAMs (surroundings). All these calculations/lookups needed to happen in a manner that would not interfere with drawing. Thus, our final design used a finite state machine (FSM).

The first stage of the FSM (INITIALIZE) initialized people into the BRAM by sending random addresses and initializing everyone with a random direction (figure 4). Their initial age was 1, and initial food count was dependent on food switch values (sw[6:0]). At the end of initialization, the BRAM signal, ppl_bram, was switched so that the drawing module could read the information that was just written to the BRAM. The random numbers used in this module were generated using a random number generator module that we designed -- which we will discuss in detail later in the report. Subsequent stages would then send addresses to the BRAM: beginning with sending the current index as the address, then sending the 8 surrounding addresses. By the fourth stage (LOOKUP_UP) we would have the information retrieved from the BRAM regarding the current index. Therefore, to save a couple clock cycles, if the information retrieved was empty, the FSM skipped to the final stage of the FSM, otherwise the current cell's

information was stored. The next stage, (LOOKUP_UPRIGHT), checked if a flood were occurring and if the current cell were within a flood zone; if so, that person would be removed from the game and the FSM transitioned to the final state. Otherwise, the FSM continued through the regular lookup process.

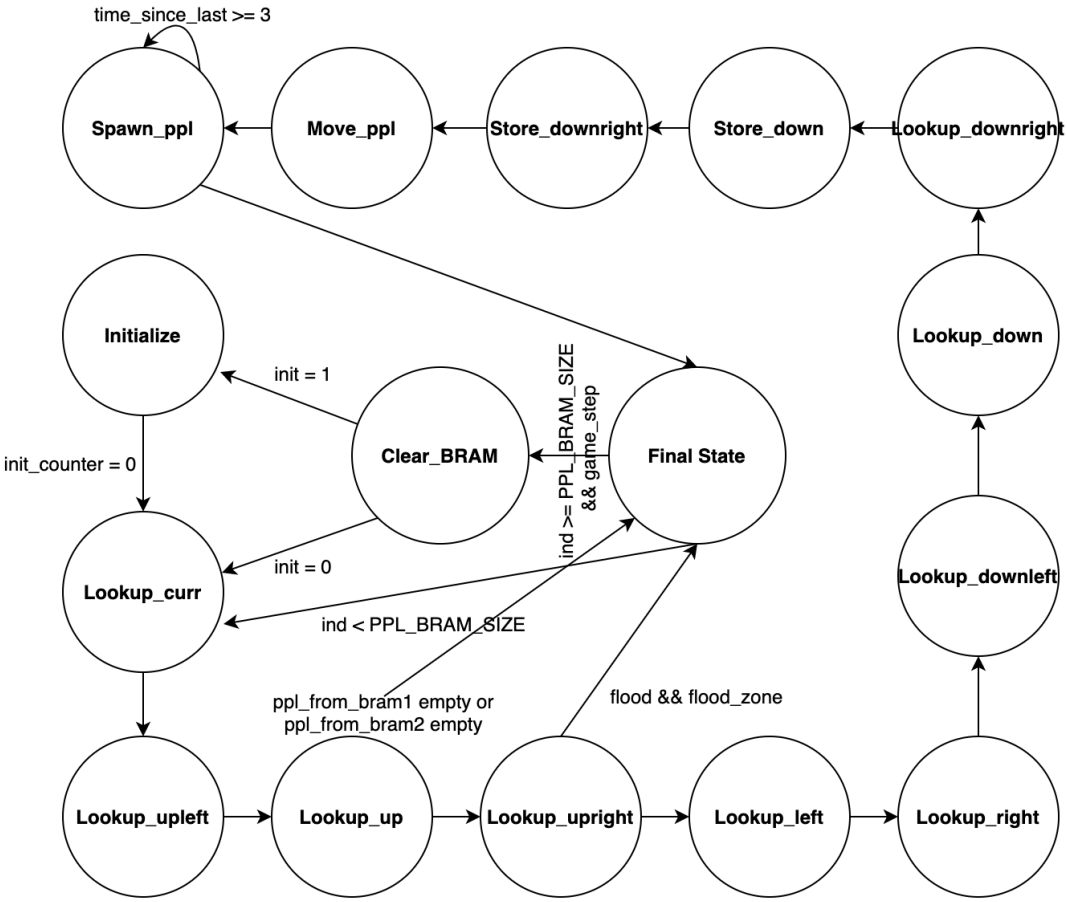

Figure 4: Diagram of the finite state machine showing the full process and logic through which each clock cycle goes through.

By the twelfth stage (MOVE_PPL), all the needed information from the BRAM was stored in the surroundings array. This stage of the FSM handled the movement and collision logic of the game. This stage first checked if the direction the person was trying to move in was currently occupied, or out of bounds; if the location was available, the person was moved by writing the new address to the BRAM and updating the age, food count, and direction of the person. Before updating the person's information, we would check: if the person's age was going to exceed the max age, if so, remove them from the game (i.e., we would write 0's to the new BRAM); if the person was in a food zone, increment that person's food count, but decrement the amount of food storage that food zone held; if the person's food count was too low, remove them from the game. In our final design, people were able to move in 3 directions (left, down and right). To increase randomization/movement, when collisions occurred, the address where the collision occurred was stored, and the two people who caused the collision were randomly

respawned elsewhere. A new person was then spawned at the location of the collision. Likewise, when a person reached the edge of the screen they would randomly respawn elsewhere -- this was an intentional design because otherwise people would get stuck at the bottom of the screen, leading to the population dying off because of decreasing number of collisions. Ultimately, to increase randomization of the game, a person's new direction was chosen randomly each game step-- with the fourth "direction" being to stay still. Additionally, we also kept track of the number of collisions each game step, if no collisions occurred within 3 game steps, we would randomly spawn 3 new people into the game -- this was because the initial location of people was based on our random number generator, so sometimes the initial locations would lead to everyone dying off very early, so this was used to prolong the game.

In the initial stages of testing, we realized that the calculations were being done at a much faster rate than the speed at which people were being drawn to the screen. Due to this, we added a signal ("game_step," which was based on a button press during testing) that slowed down the speed of the calculations. The final state of the FSM is where we would wait for the game_step signal, which only triggered after reading/writing to the new BRAM was complete. We would then flip the value of the valid ("ppl_bram") signal -- for which BRAM, the drawing module would read from in the next game step. We also kept track of a counter that would replenish the amount of food that food zones held -- this was meant to make the food zones a little more realistic; if populations ate the food too quickly, they would die out very quickly. Lastly, when a game step occurred, we would first clear the BRAM we were going to write to, so that no old information would carry over in between game steps, then proceed to write new information based on the now "old" BRAM.

The bulk of our project went into designing (and re-designing) this module, since it handled the entirety of the game logic. Usually any issues/bugs we came across in the early stages of the project stemmed from this module. Thus, the majority of our time went into using multiple ILAs to debug and determine which parts of the module did not function properly. Thankfully, due to the separation in states of the FSM we were able to debug rather efficiently, besides the issues that were due to timing i.e., the timing delay when drawing.

3.3 Drawing (Mario)

To have all the graphics show up, the drawing logic was dispersed between three submodules sending data to the main drawing module. The drawing module receives all its inputs from top level and sends these to the submodules which do most of the work in creating the RGB pixel output for the screen and sends it back to top level. All pixel values of zero are converted to yellow before being sent to top level, indicating that it is part of the island; this is done to prevent colors from mixing. Drawing also calculates the current BRAM address to read from by considering hcount and vcount and converting them into grid spaces using

$$address = ISLAND\_WIDTH \times grid\_y + grid\_x, \tag{1}$$

where

$$grid\_x = \frac{hcount - LEFT\_EDGE}{8} \quad \text{and} \quad grid\_y = \frac{vcount - TOP\_EDGE}{8}.$$

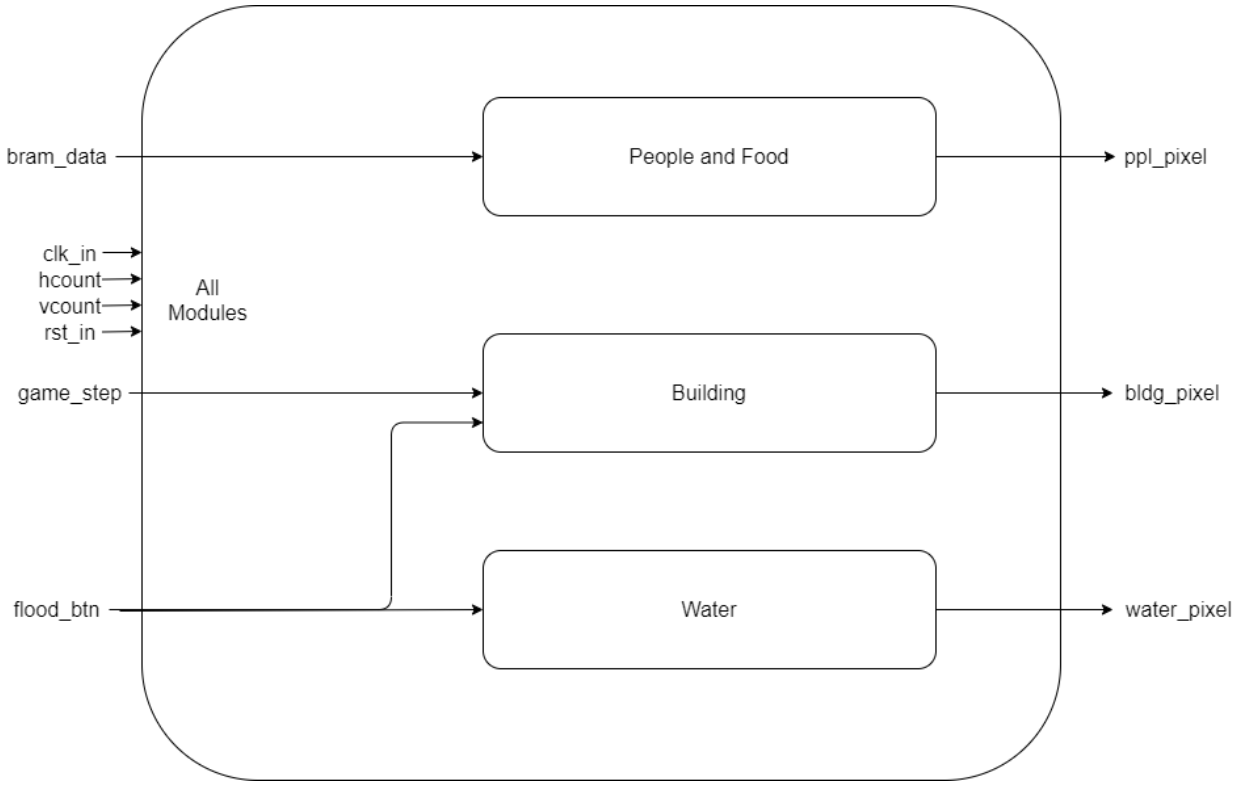

Figure 5: Diagram showing the inputs and outputs of the submodules of Drawing.

3.3.1 People and Food

To draw people, food, and the island without overlapping colors, the people and food module prioritizes drawing people using the data from the BRAM. This module checks to see if the 22 people bits in the BRAM are greater than zero, indicating that there is a person occupying the grid space. If this is greater than zero, then the pixel color gets set to red. If there is not a person there, then it checks to see if the grid space is a food zone. There are two food zones with set locations. The module checks to see if hcount and vcount are within the predetermined regions, and if they are, then the pixel color gets set to green. If a grid space is neither a person nor food zone, then the pixel color is set to zero indicating it is part of the island. The module prioritizes people being drawn to prevent any color overlapping which may occur between the three colors on the island. The current pixel color is always being sent to the drawing module for it to output the right color on screen.

3.3.2 Building

The building module uses the idea of a grid by creating a 24-bit array where each bit represents a building in a 12x2 grid. These buildings occupied the top and bottom rows of the island with each building being a size of 64x96 pixels. To create this grid, the building module was given game_step as a parameter, which was fed through the drawing module from top level. Game_step indicated one step in time of the game, meaning one movement per person on the screen. This was used to increment a counter in the building module that counted to 31 game cycles. When this counter was at its max, the module would insert a new building into the grid

wherever it found the first zero in the array. Each instance of 1 in the 24-bit array indicated that a building had been built in that location on the grid. One of our goals was to create buildings due to the groupings of people on screen; however, due to time constraints and the limitations of the functionality of checking surroundings, this was unable to be implemented. To make sure that a flood destroyed the buildings which ended up under water, there was a case to check if the flood button was pressed and if so, then the grid locations (0, 1, 10, 11, 12, 13, 22 and 23) which were meant to be under water were cleared to 0.

After the building grid was generated, the first thing that the building module would do is check to see if the current grid space contained a building. To iterate through all the indices in the grid, hcount was used to determine the current index. If the current index contained a building, then it would check to see if it was in the 0 to 11 range, meaning that it was on the top row of the island. If this was the case, then it would check if vcount was in the correct range, between 0 and 96 pixels. If this was the case, then the pixel color gets set to one of two shades of brown. The color of brown is determined by checking the modulus of the index with 2. If the modulus was equivalent to 0, then the darker shade of brown was set to the pixel color, this was to make buildings distinguishable from one another and have alternating color patterns. The process above was repeated for the lower row, this time checking to see if the index was between 12 and 23 and vcount was between 672 and 768 pixels. Equation number was also used to determine the hcount range. If the current pixel was not a building, then it was set to zero.

3.3.3 Water

Water essentially has two drawing states. One state is static which always draws water on the left and right sides of the island, always 128 pixels wide. The second state is the flood state, where the water size increases to 256 on both sides and overlaps with the island. This module continuously checks to see if the pixel is in the water range which is always the first and last hcount pixels on the screen but increases to 256 pixels when the flood button is pressed.

3.4 Random Number Generator (RNG) *(Albert)*

We designed a random number generator by creating two linear-feedback shift registers (LFSRs). One was 32-bits, and the other was 31-bits. LFSRs are generated by left shifting the initial seed value by 1, and the final bit is the result of using XOR on the first bit with the tap value (the tap value is just which bit we XOR with the first bit, different tap values start with the same initial random values, but eventually diverge). The initial seed value for the 31-bit LFSR was always the same, however the initial seed value for the 32-bit LFSR was based on switch values. This allowed us to try out different seed values -- some seed values are better than others, so some would lead to populations lasting longer than others. To further increase randomization, we used XOR on the two LFSRs with one another and returned that.

We used the RNG when choosing random locations at which to spawn people, and when randomly choosing the initial direction and new direction each game step. Since the randomly generated number was 32-bits long we truncated it to fit our needs. This created enough pseudo randomness that was suitable enough for our game.

4 Testing and Debugging (Nailah)

For testing, we used a combination of ILAs, testbenches, and slowing down the run time on the screen to check each state of the game. As mentioned before, game_step was initially a button press so that we could see step-by-step what was happening in the simulation. This was

helpful for both screen view and ILA testing. The ILA was most helpful in later stages of the project, while testbenches were used to double check our early modules and make sure everything was functioning as expected. A testbench for the RNG module was used to debug and confirm that our numbers generated were random enough. We also used a test bench for the early stages of the People and Food module, checking that different hcount, vcount values would output the correct RGB number.

When debugging Movement & Collision, it was helpful to deliberately place a person on the screen instead of using the RNG module. This way, the starting address of the person was known, and, using an ILA trigger on that index location, we could check and see if the surroundings and other information at that index were correct. Several iterations of the ILA were made to check values such as lookup_addr, surroundings, and information being sent to and retrieved from the BRAM. The ILA was one of the most important tools we used in debugging, and how we ultimately got people to move in the simulation.

5 Challenges

In our initial design, we relied on modulo operations to determine if indices were within a certain range (mainly to check if people were on the left or right edge of the screen). However, after doing some testing, we realized that modulo operations were only supported in simulations; modulo operations are not supported in System Verilog unless they are a power of 2, so we had to implement our own modulo function. This proved to be a challenge in the beginning because we failed to realize that the modulo operations -- or rather, the inability to perform modulo operations -- were throwing off our calculations.

We also realized during our design process that the addresses at which we were storing information did not align with drawing's timing process. When trying to keep a person at the same index as the previous game_step (i.e., keep the person from moving), in the next game step, the person would have moved to the cell down-right from its previous position. Our best guess was that this was a result of screen timing as people were drawn onto the screen, though we were not completely sure this was the explanation. Either way, to fix the issue, we had to offset the addresses at which the people were stored. To stay still, the people were stored at a new address, which equaled the current index plus GRID_WIDTH (to offset the down direction) minus 1 (to offset the right direction). The three directions of movement--left, right, and down--were modified accordingly. We attempted to also fix the up direction, but no matter what we changed, the person would still only move down, which is what led us to believe that the issue was a result of screen timing. This ultimately led to only being able to implement 3 directions. As a result, we decided to randomly respawn people when they reached the edge of the screen otherwise, they would get stuck at the bottom of the screen -- the decision to randomly respawn people was due to glitching occurring when we attempted to wrap people around the edge of the screen i.e., we tried to have them pop up on the opposite side of the screen.

Below: images of the failed simulation when we tried to use all directions

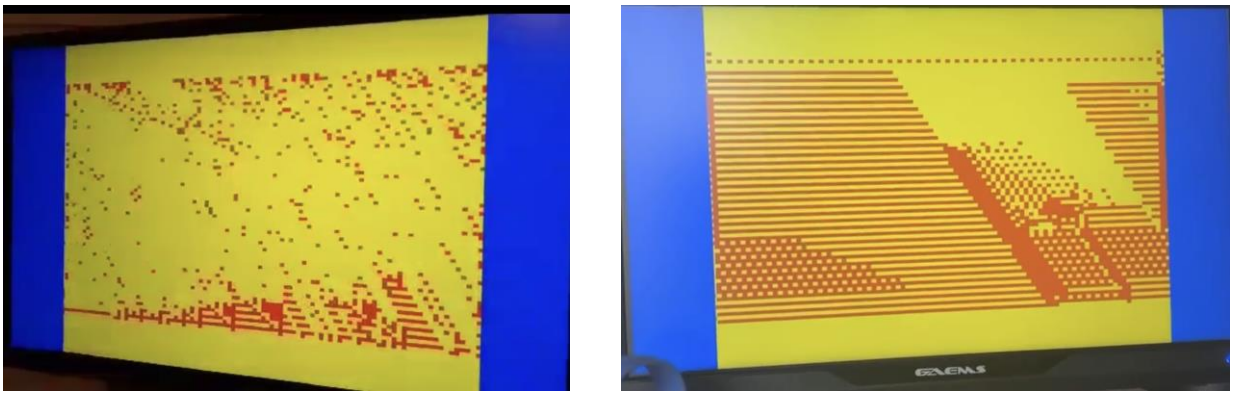

Figure 6: Failed simulations when using all directions.

Another issue that stemmed from screen timing issues was our surroundings/lookup_addr arrays. This was because we had to offset the indices when trying to keep people at the same position for the next game step. Because of this, we did not know whether the addresses we were looking up were the correct ones or if we had to offset them. When we tried to use an ILA to check whether the retrieved information was correct, we were unable to get any information back from the BRAM, which led to even more confusion. Ultimately, we determined that our lookup_addr was correct, but for some reason our surroundings array was incorrect. The malfunctioning of the surroundings led to glitching of our game when trying to move people around the screen, which is what led to us randomly respawning people when they hit the edge of the screen.

A related problem to the surroundings issue was our implementation of food zones. Originally, we had many 2x2 food zones spread around the island, placed at random just like the people were. In this version, when a person moved, we needed to make sure that the information at this grid space would keep the same food information, i.e., the same food_indicator and food_capacity bits. To do this, when needed to use the surroundings information to keep the food info of the cell that the person was moving too. However, because surroundings did not work as expected, this resulted in deformed food zones, meaning they had lost grid spaces, added grid spaces, or both. This meant that we could not see an accurate depiction of where the food zones were, which is why we ultimately chose to use two large food zones. This way, drawing would always draw the zones in the same place, and we could just store the zone's information in two separate arrays.

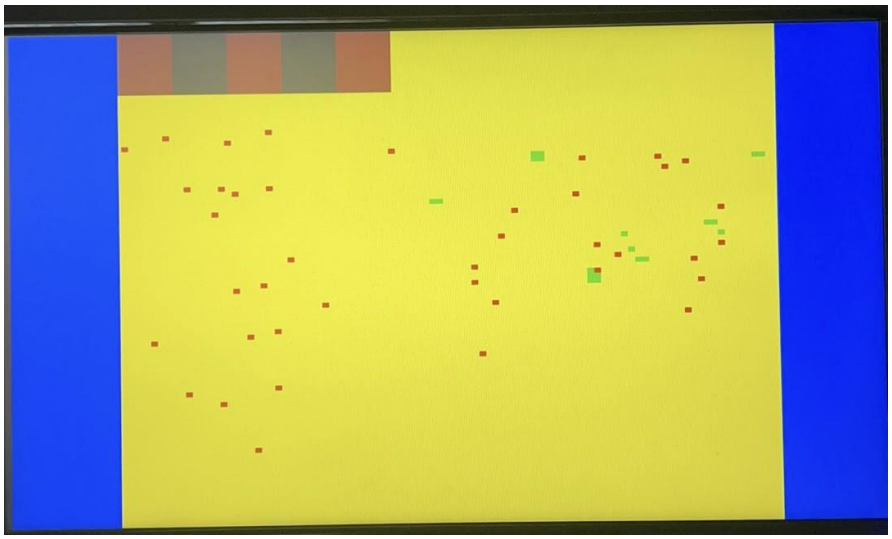

Figure 7: Simulation with deformed and missing food zones.

6 Conclusion

While the project may have not turned out as expected, most of our minimum goals were achieved. People were able to move in different directions across the screen with correct collision logic. Flooding the island correctly removes people and buildings in the affected area. The random number generator functions properly and was implemented throughout the project. Food zones were not functional, and the earthquake module was never created due to the many hours put into figuring out how to move people in the correct direction. Even with some aspects of the game left out, it is still fully functional and fun to see the generation of people over time.

7 Future Suggestions

If we had a second chance at this project, one of the things we would change in our design is to send hcount and vcount to both Drawing and Movement & Collision. In the final version of this project, we check for flood zones and food zones by using combinational logic but checking all these index ranges was inconvenient and not the best method to use, especially if we wanted to move our food zones or flood areas. By sending hcount and vcount to Movement & Collision, we would have a lot fewer ranges to check, since we would have both the x and y value of that pixel (or the grid space if we divide by 8), rather than just the converted address stored in the BRAM.

Another suggestion is to put more time in the planning part of the project into the game FSM. The bulk of the logic for this FSM was in the MOVE_PPL state, and there were so many steps that needed to happen that this could have benefited from a well-defined minor FSM. We think that the reason we were not able to get food count working was because we were implementing steps in the wrong order, so planning this out ahead of time could have mitigated the number of errors we ran into.

Acknowledgements

We would like to thank Joe Steinmeyer for helping us get through all the challenges that we faced. We would also like to thank the everyone on the 6.111 staff team who helped us learn Verilog and debug our code.

References

[1] Steinmeyer, J., "Lab 3" 6.111 Laboratory Instructions, MIT [Unpublished Online]. Available: https://6111.io/F20/labs/lab03. [Accessed: 10-Nov-2020].

[2] "Random Number Generation Using LFSR | Maxim Integrated" [Online]. Available: https://www.maximintegrated.com/en/design/technical-documents/app-notes/4/4400.html. [Accessed: 13-Dec-2020].

Code Appendix

Top Level

```
1.  module top_level(
2.     input clk_100mhz,
3.     input[15:0] sw,
4.     input btnc, btnu, btnd,
5.     output logic[3:0] vga_r,
6.     output logic[3:0] vga_b,
7.     output logic[3:0] vga_g,
8.     output logic vga_hs,
9.     output logic vga_vs
10.    );
11.     clk_wiz_0 clkdivider(.clk_in1(clk_100mhz), .clk_out1(clk_65mhz), .clk_out2(clock_100mhz),
    .reset(btnd));
12.
13.     logic [10:0] hcount;    // pixel on current line
14.     logic [9:0] vcount;     // line number
15.     logic hsync, vsync;
16.     logic [11:0] pixel;
17.
18.     xvga xvga1(.vclock_in(clk_65mhz),.hcount_out(hcount),.vcount_out(vcount),
19.           .hsync_out(hsync),.vsync_out(vsync),.blank_out(blank));
20.
21.     logic clean_btnu, old_clean_btnu;
22.     debounce db1(.reset_in(btnd), .clock_in(clk_65mhz), .noisy_in(btnu), .clean_out(clean_btnu));
23.     always_ff @(posedge clk_65mhz) old_clean_btnu <= clean_btnu;
24.
25.     logic clean_flood, old_clean_flood;
26.     debounce db2(.reset_in(btnd), .clock_in(clk_65mhz), .noisy_in(btnc), .clean_out(clean_flood));
27.     always_ff @(posedge clk_65mhz) old_clean_flood <= clean_flood;
28.
29.     logic build, ppl_bram;
30.     //logic [2:0] ppl_bram_buff;
31.     logic [12:0] addrb;
32.     logic [29:0] data_out_draw1, data_out_draw2, data_to_draw;
33.     assign data_to_draw = (ppl_bram) ? data_out_draw1 : data_out_draw2; //might need to change this
34.
35.     logic game_step;
36.
37.     //always_ff @(posedge clk_65mhz) ppl_bram_buff <= {ppl_bram_buff[1:0], ppl_bram}; //to safely send
    ppl_bram across clock domains
38.
39.     drawing display(.clk_in(clk_65mhz), .rst_in(btnd),
    .hsync(hsync),.vsync(vsync),.blank(blank),.hcount(hcount),
40.          .vcount(vcount), .vga_r(vga_r),.vga_b(vga_b),.vga_g(vga_g),.vga_hs(vga_hs),.vga_vs(vga_vs),
41.          .flood(btnc), .bram_data(data_to_draw), .address(addrb), .game_step(game_step),
    .build(build));
42.
43.     logic [9:0] food_switches;
44.     assign food_switches = sw[9:0];
45.     logic [2:0] rng_sel;
46.     assign rng_sel = sw[15:13];
47.
48.     logic wea1, wea2;
49.     logic [12:0] addra_1, addra_2;
50.     logic [29:0] ppl_in_write, data_out1, data_out2;
51.     movement_and_collision game_log(.rst_in(btnd), .clk_in(clk_65mhz), .food_sw(food_switches),
    .flood(clean_flood && ~old_clean_flood),
52.          .write_ppl_1(wea1), .write_ppl_2(wea2), .ppl_to_bram(ppl_in_write), .addr_ppl_1(addra_1),
    .rng_sel(rng_sel), .build(build),
53.          .addr_ppl_2(addra_2), .ppl_from_bram_1(data_out1), .ppl_from_bram_2(data_out2),
    .ppl_bram(ppl_bram), .game_step(game_step));
54.
55.     people_BRAM people1(.clka(clk_65mhz), .ena(1), .wea(wea1), .addra(addra_1), .dina(ppl_in_write),
    .douta(data_out1),
56.          .clkb(clk_65mhz), .enb(1), .web(0), .addrb(addrb), .dinb(19'b0), .doutb(data_out_draw1));
57.
58.     people_BRAM people2(.clka(clk_65mhz), .ena(1), .wea(wea2), .addra(addra_2), .dina(ppl_in_write),
    .douta(data_out2),
59.          .clkb(clk_65mhz), .enb(1), .web(0), .addrb(addrb), .dinb(19'b0), .doutb(data_out_draw2));
60.
61.     logic [27:0] counter;
62.     parameter CYCLES = 27'd10000000;
63.     always_ff @(posedge clk_65mhz) begin
64.         if (btnd) begin
65.             counter <= 0;
66.         end else begin
```

```
            counter <= counter + 1;
            if (counter == CYCLES) begin
                counter <= 0;
                game_step <= 1;
            end else begin
                game_step <= 0;
            end
        end
    end

endmodule //top_level
```

xvga

```
module xvga(input vclock_in,
            output logic [10:0] hcount_out,    // pixel number on current line
            output logic [9:0] vcount_out,     // line number
            output logic vsync_out, hsync_out,
            output logic blank_out);

    parameter DISPLAY_WIDTH  = 1024;      // display width
    parameter DISPLAY_HEIGHT = 768;       // number of lines

    parameter  H_FP = 24;                 // horizontal front porch
    parameter  H_SYNC_PULSE = 136;        // horizontal sync
    parameter  H_BP = 160;                // horizontal back porch

    parameter  V_FP = 3;                  // vertical front porch
    parameter  V_SYNC_PULSE = 6;          // vertical sync
    parameter  V_BP = 29;                 // vertical back porch

    // horizontal: 1344 pixels total
    // display 1024 pixels per line
    logic hblank,vblank;
    logic hsyncon,hsyncoff,hreset,hblankon;
    assign hblankon = (hcount_out == (DISPLAY_WIDTH -1));
    assign hsyncon = (hcount_out == (DISPLAY_WIDTH + H_FP - 1));  //1047
    assign hsyncoff = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE - 1));  // 1183
    assign hreset = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE + H_BP - 1));  //1343

    // vertical: 806 lines total
    // display 768 lines
    logic vsyncon,vsyncoff,vreset,vblankon;
    assign vblankon = hreset & (vcount_out == (DISPLAY_HEIGHT - 1));   // 767
    assign vsyncon = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP - 1));  // 771
    assign vsyncoff = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE - 1));  // 777
    assign vreset = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE + V_BP - 1)); // 805

    // sync and blanking
    logic next_hblank,next_vblank;
    assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
    assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
    always_ff @(posedge vclock_in) begin
       hcount_out <= hreset ? 0 : hcount_out + 1;
       hblank <= next_hblank;
       hsync_out <= hsyncon ? 0 : hsyncoff ? 1 : hsync_out;  // active low

       vcount_out <= hreset ? (vreset ? 0 : vcount_out + 1) : vcount_out;
       vblank <= next_vblank;
       vsync_out <= vsyncon ? 0 : vsyncoff ? 1 : vsync_out;  // active low

       blank_out <= next_vblank | (next_hblank & ~hreset);
    end
endmodule //xvga
```

Movement and Collision

```
module movement_and_collision(
     input clk_in, //100mhz clock
     input [2:0] rng_sel, //switches to determine rng initial seed
     input [9:0] food_sw, //food count determined by user input
     input flood,
     input rst_in,
     input game_step,
     output logic write_ppl_1, //wea signal for people BRAM
     output logic write_ppl_2,
```

```
10.     output logic [12:0] addr_ppl_1, //address of the people we insert into people BRAM, will be used
    for lookups (13 bits)
11.     output logic [12:0] addr_ppl_2,
12.     output logic [29:0] ppl_to_bram, //data sent to people BRAM {counter (5 bits), food_count (7bits),
    age (8 bits), dir (2 bits)} (30 bits)
13.     input [29:0] ppl_from_bram_1,
14.     input [29:0] ppl_from_bram_2,
15.     output logic ppl_bram, //tells us which BRAM we are reading from/writing to
16.     output logic build
17.     );
18.     localparam MAX = 13'd8190;
19.     localparam GRID_WIDTH = 13'd96;
20.     localparam GRID_HEIGHT = 13'd72;
21.     localparam PPL_BRAM_SIZE = GRID_WIDTH*GRID_HEIGHT - 1;
22.
23.     //game parameters
24.     localparam MAX_AGE = 10'd127;
25.     localparam init_age = 10'b1;
26.     logic [26:0] initial_ppl_state;
27.     logic [12:0] ind; //used for indexing into grid
28.     logic [12:0] ppl_counter;
29.
30.     logic [9:0] food_storage1;
31.     logic [9:0] food_storage2;
32.     logic [4:0] repl_food;
33.
34.     //storing parameters
35.     logic [9:0] food_count;
36.     logic [9:0] age;
37.     //logic [4:0] counter; //will implement later
38.     logic [1:0] dir;
39.
40.     //for collisions and randomly spawning ppl
41.     logic [2:0] num_collisions;
42.     logic [1:0] time_since_last;
43.     logic [12:0] stored_addr;
44.     logic init;
45.
46.     logic [9:0] init_food;
47.     assign initial_ppl_state = {8'b0, init_food, init_age}; //initialization of ppl is dependent on
    food sw values, leave dir empty for initialization
48.
49.     //game states
50.     logic [4:0] game_state;
51.     logic [5:0] init_counter; //used to initialize ppl & food into bram
52.     logic [12:0] lookup_addr [7:0]; //eight surrounding locations
53.     logic [29:0] surroundings [7:0];//data from surrounding cells
54.
55.     //game states
56.     localparam INITIALIZE = 5'b0000;
57.     localparam LOOKUP_CURR = 5'b0001;
58.     localparam LOOKUP_UPLEFT = 5'b0010;
59.     localparam LOOKUP_UP = 5'b0011;
60.     localparam LOOKUP_UPRIGHT = 5'b0100;
61.     localparam LOOKUP_LEFT = 5'b0101;
62.     localparam LOOKUP_RIGHT = 5'b0110;
63.     localparam LOOKUP_DOWNLEFT = 5'b0111;
64.     localparam LOOKUP_DOWN = 5'b1000;
65.     localparam LOOKUP_DOWNRIGHT = 5'b1001;
66.     localparam STORE_DOWN = 5'b1010;
67.     localparam STORE_DOWNRIGHT = 5'b1011;
68.     localparam MOVE_PPL = 5'b1100;
69.     localparam SPAWN_PPL = 5'b1101;
70.     localparam STORE_FOOD = 5'b1110;
71.     localparam FINAL_STATE = 5'b1111;
72.     localparam CLEAR_BRAM = 5'b10000;
73.     //add replenish food state
74.
75.     //fsm states for direction:
76.     //localparam UP = 3'b000;
77.     //localparam UP_RIGHT = 3'b001;
78.     localparam RIGHT = 2'b10;
79.     //localparam DOWN_RIGHT = 3'b011;
80.     localparam DOWN = 2'b00;
81.     //localparam DOWN_LEFT = 3'b101;
82.     localparam LEFT = 2'b01;
83.     //localparam UP_LEFT = 3'b111;
84.
85.     logic [31:0] random_num;
86.     rng random_num_gen(.rst_in(rst_in), .clk_in(clk_in), .num_sel(rng_sel), .shifted_res(random_num));
87.
```

```
88.     logic [1:0] mod;
89.     modulus mod_res(.ind(ind), .mod(mod)); //if mod=0:ind%GRID_WIDTH=0, mod = 1:ind%GRID_WIDTH=95,
    2=neither
90.
91.     logic [29:0] data_from_bram_1;
92.     logic [12:0] addr_1, addr_2;
93.     logic which_bram;
94.     assign addr_1 = addr_ppl_1;
95.     assign addr_2 = addr_ppl_2;
96.     assign which_bram = ppl_bram;
97.     assign data_from_bram_1 = ppl_from_bram_1;
98.     //ila_1 myila(.clk(clk_in), .probe0(which_bram), .probe1(game_state), .probe2(ind), .probe3({4'b0,
    addr_1, addr_2}), .probe4(data_from_bram_1));
99.
100.    logic flood_zone;
101.    assign flood_zone =
    (ind>=0&&ind<16)||(ind>=80&&ind<112)||(ind>=176&&ind<208)||(ind>=272&&ind<304)||(ind>=368&&ind<400)||
102.                        (ind>=464&&ind<496)
    ||(ind>=560&&ind<592)||(ind>=656&&ind<688)||(ind>=752&&ind<784)||(ind>=848&&ind<880)||
103.
    (ind>=944&&ind<976)||(ind>=1040&&ind<1072)||(ind>=1136&&ind<1168)||(ind>=1232&&ind<1264)||(ind>=1328&&
    ind<1360)||
104.
    (ind>=1424&&ind<1456)||(ind>=1552&&ind<1584)||(ind>=1616&&ind<1648)||(ind>=1712&&ind<1744)||(ind>=1808
    &&ind<1840)||
105.
    (ind>=1904&&ind<1936)||(ind>=2000&&ind<2032)||(ind>=2096&&ind<2128)||(ind<=2192&&ind<2224)||(ind>=2288
    &&ind<2320)||
106.
    (ind>=2384&&ind<2416)||(ind>=2480&&ind<2512)||(ind>=2576&&ind<2608)||(ind>=2672&&ind<2704)||(ind>=2768
    &&ind<2800)||
107.
    (ind>=2864&&ind<2896)||(ind>=2960&&ind<2992)||(ind>=3056&&ind<3088)||(ind>=3152&&ind<3184)||(ind>=3248
    &&ind<3280)||
108.
    (ind>=3344&&ind<2276)||(ind>=3440&&ind<3472)||(ind>=3536&&ind<3568)||(ind>=3632&&ind<3664)||(ind>=3728
    &&ind<3760)||
109.
    (ind>=3824&&ind<3856)||(ind>=3920&&ind<3952)||(ind>=4016&&ind<4048)||(ind>=4112&&ind<4144)||(ind>=4208
    &&ind<4240)||
110.
    (ind>=4304&&ind<4336)||(ind>=4400&&ind<4432)||(ind>=4496&&ind<4528)||(ind>=4592&&ind<4624)||(ind>=4688
    &&ind<4720)||
111.
    (ind>=4784&&ind<4816)||(ind>=4880&&ind<4912)||(ind>=4976&&ind<5008)||(ind>=5072&&ind<5104)||(ind>=5168
    &&ind<5200)||
112.
    (ind>=5264&&ind<5296)||(ind>=5360&&ind<5392)||(ind>=5456&&ind<5488)||(ind>=5552&&ind<5584)||(ind>=5648
    &&ind<5680)||
113.                        (ind>=5744&&ind<5776)||(ind>=
    5840&&ind<5872)||(ind>=5936&&ind<5968)||(ind>=6032&&ind<6064)||(ind>=6128&&ind<6160)||
114.
    (ind>=6224&&ind<6256)||(ind>=6320&&ind<6352)||(ind>=6416&&ind<6448)||(ind>=6512&&ind<6544)||(ind>=6608
    &&ind<6640)||
115.                        (ind>=6704&&ind<6736)||(ind>=6800&&ind<6832)||(ind>=6896) ? 1 : 0;
116.    logic food_zone1;
117.    assign food_zone1 =
    (ind>=1168&&ind<1184)||(ind>=1264&&ind<1280)||(ind>=1360&&ind<1376)||(ind>=1456&&ind<1472)||(ind>=1552
    &&ind<1568)||
118.
    (ind>=1648&&ind<1664)||(ind>=1744&&ind<1760)||(ind>=1840&&ind<1856)||(ind>=1936&&ind<1952)||(ind>=2032
    &&ind<2048)||
119.
    (ind>=2128&&ind<2144)||(ind>=2224&&ind<2240)||(ind>=2320&&ind<2336)||(ind>=2416&&ind<2432)||(ind>=2512
    &&ind<2528)||
120.
    (ind>=2608&&ind<2624)||(ind>=2704&&ind<2720)||(ind>=2800&&ind<2816)||(ind>=2896&&ind<2912)||(ind>=2992
    &&ind<3008)||
121.
    (ind>=3088&&ind<3104)||(ind>=3184&&ind<3200)||(ind>=3280&&ind<3296)||(ind>=3376&&ind<3392)||(ind>=3472
    &&ind<3488)||
122.
    (ind>=3568&&ind<3584)||(ind>=3664&&ind<3680)||(ind>=3760&&ind<3776)||(ind>=3856&&ind<3872)||(ind>=3952
    &&ind<3968)||
123.
    (ind>=4048&&ind<4064)||(ind>=4144&&ind<4160)||(ind>=4240&&ind<4256)||(ind>=4336&&ind<4352)||(ind>=4432
    &&ind<4448)||
124.
    (ind>=4528&&ind<4544)||(ind>=4624&&ind<4640)||(ind>=4720&&ind<4736)||(ind>=4816&&ind<4832)||(ind>=4912
    &&ind<4928)||
```

```
125.
    (ind>=5008&&ind<5024)||(ind>=5104&&ind<5120)||(ind>=5200&&ind<5216)||(ind>=5296&&ind<5312)||(ind>=5392
    &&ind<5408)||
126.
    (ind>=5488&&ind<5504)||(ind>=5584&&ind<5600)||(ind>=5680&&ind<5696)||(ind>=5776&&ind<5792) ? 1 : 0;
127.
128.    logic food_zone2;
129.    assign food_zone2 =
    (ind>=80&&ind<96)||(ind>=176&&ind<192)||(ind>=272&&ind<288)||(ind>=368&&ind<384)||(ind>=464&&ind<480)||
    |
130.
    (ind>=560&&ind<576)||(ind>=656&&ind<672)||(ind>=752&&ind<768)||(ind>=848&&ind<864)||(ind>=944&&ind<960
    )||
131.
    (ind>=1040&&ind<1056)||(ind>=1136&&ind<1152)||(ind>=1232&&ind<1248)||(ind>=1328&&ind<1344)||(ind>=1424
    &&ind<1440)||
132.
    (ind>=1520&&ind<1536)||(ind>=1616&&ind<1632)||(ind>=1712&&ind<1728)||(ind>=1808&&ind<1824)||(ind>=1904
    &&ind<1920)||
133.
    (ind>=2000&&ind<2016)||(ind>=2096&&ind<2112)||(ind>=2192&&ind<2208)||(ind>=2288&&ind<2304)||(ind>=2384
    &&ind<2400)||
134.
    (ind>=2480&&ind<2496)||(ind>=2576&&ind<2592)||(ind>=2672&&ind<2688)||(ind>=2768&&ind<2784)||(ind>=2864
    &&ind<2880)||
135.
    (ind>=2960&&ind<2976)||(ind>=3056&&ind<3072)||(ind>=3152&&ind<3168)||(ind>=3248&&ind<3264)||(ind>=3344
    &&ind<3360)||
136.
    (ind>=3440&&ind<3456)||(ind>=3536&&ind<3552)||(ind>=3632&&ind<3648)||(ind>=3728&&ind<3744)||(ind>=3824
    &&ind<3840)||
137.
    (ind>=3920&&ind<3936)||(ind>=4016&&ind<4032)||(ind>=4112&&ind<4128)||(ind>=4208&&ind<4224)||(ind>=4304
    &&ind<4320)||
138.                        (ind>=4400&&ind<4416)||(ind>=4496&&ind<4512)||(ind>=4592&&ind<4608) ? 1 : 0;
139.
140.    always_comb begin
141.      lookup_addr[0] = (ind<GRID_WIDTH) || (mod==0) ? MAX: ind - GRID_WIDTH - 1; //upper left
142.      lookup_addr[1] = (ind<GRID_WIDTH) ? MAX: ind - GRID_WIDTH; //direct above
143.      lookup_addr[2] = (ind<GRID_WIDTH) || (mod==1) ? MAX: ind - GRID_WIDTH + 1; //upper right
144.      lookup_addr[3] = (mod==0)? MAX: ind - 1; //direct left
145.      lookup_addr[4] = (mod==1) ? MAX: ind + 1; //direct right
146.      lookup_addr[5] = (ind>=6816) || (mod==0) ? MAX: ind + GRID_WIDTH - 1; //lower left
147.      lookup_addr[6] = (ind>=6816) ? MAX: ind + GRID_WIDTH; //direct below
148.      lookup_addr[7] = (ind>=6816) || (mod==1) ? MAX: ind + GRID_WIDTH + 1; //lower right
149.    end
150.
151.     always_ff @(posedge clk_in) begin
152.        case (game_state)
153.          INITIALIZE: begin
154.                    init_counter <= init_counter - 1;
155.                    if (init_counter == 0) begin
156.                        //set everything to 0 and start game
157.                        write_ppl_1 <= 0;
158.                        write_ppl_2 <= 0;
159.                        addr_ppl_1 <= 0;
160.                        ppl_to_bram <= 0;
161.                        init <= 0;
162.                        ppl_bram <= ~ppl_bram;
163.                        game_state <= LOOKUP_CURR;
164.                    end else if (init_counter == 6'd50) begin
165.                        //initializing max boundary
166.                        write_ppl_1 <= 1;
167.                        write_ppl_2 <= 1;
168.                        addr_ppl_1 <= MAX;
169.                        addr_ppl_2 <= MAX;
170.                        ppl_to_bram <= 30'h7FFFF003;
171.                    end else if (init_counter <= 6'd8 && init_counter > 0) begin
172.                        //spawning ppl
173.                        write_ppl_1 <= 1;
174.                        addr_ppl_1 <= random_num[11:0];
175.                        ppl_to_bram <= {initial_ppl_state, random_num[1:0]};
176.                    end
177.                end
178.          LOOKUP_CURR: begin //lookup using count
179.                     write_ppl_1 <= 0;
180.                     write_ppl_2 <= 0;
181.                     if (ppl_bram) addr_ppl_1 <= ind;
182.                     else addr_ppl_2 <= ind;
183.                     game_state <= game_state + 1;
184.                   end
```

```
185.          LOOKUP_UPLEFT:begin //feed in addr to get upper left cell
186.                    if (ppl_bram) addr_ppl_1 <= lookup_addr[0];//if A is ative get from A
187.                    else addr_ppl_2 <= lookup_addr[0]; //else get from B
188.                    game_state <= game_state + 1;
189.                end
190.          LOOKUP_UP:begin
191.                    if (ppl_bram) addr_ppl_1 <= lookup_addr[1]; //ask for upper one from A
192.                    else addr_ppl_2 <= lookup_addr[1]; //or ask for upper one from B
193.                    //by this point we have center point cell data
194.
195.                    if (ppl_bram) begin //if empty jump to final state
196.                        if (ppl_from_bram_1[11:2] == 0) game_state <= FINAL_STATE;
197.                        else begin //store the current cell's information
198.                            //counter <= (ppl_bram) ? ppl_from_bram_1[21:17] : ppl_from_bram_2[21:17];
    //5
199.                            food_count <= ppl_from_bram_1[21:12]; //10
200.                            age <= ppl_from_bram_1[11:2]; //10
201.                            dir <= ppl_from_bram_1[1:0]; //2
202.                            game_state <= game_state + 1;
203.                        end
204.                    end else if (~ppl_bram) begin
205.                        if (ppl_from_bram_2[11:2] == 0) game_state <= FINAL_STATE;
206.                        else begin //store the current cell's information
207.                            //counter <= (ppl_bram) ? ppl_from_bram_1[21:17] : ppl_from_bram_2[21:17];
    //5
208.                            food_count <= ppl_from_bram_2[21:12]; //7
209.                            age <= ppl_from_bram_2[11:2]; //8
210.                            dir <= ppl_from_bram_2[1:0]; //2
211.                            game_state <= game_state + 1;
212.                        end
213.                    end
214.                end
215.          LOOKUP_UPRIGHT:begin //beginning here we need to feed in new address and read out the output
216.                    if (flood) begin //check if flood
217.                        if (flood_zone) begin
218.                            game_state <= FINAL_STATE;
219.                            if (~ppl_bram) begin
220.                                addr_ppl_1 <= ind;
221.                                write_ppl_1 <= 1;
222.                                ppl_to_bram <= 30'b0;
223.                            end else begin
224.                                addr_ppl_2 <= ind;
225.                                write_ppl_2 <= 1;
226.                                ppl_to_bram <= 30'b0;
227.                            end
228.                        end else begin
229.                            write_ppl_1 <= 0;
230.                            write_ppl_2 <= 0;
231.
232.                            if (ppl_bram) addr_ppl_1 <= lookup_addr[2]; //ask for upper right from A
233.                            else addr_ppl_2 <= lookup_addr[2];  //ask for upper right from B
234.
235.
236.                            if (ppl_bram) surroundings[0] <= ppl_from_bram_1; //data present from
    upper left lookup. store it
237.                            else surroundings[0] <= ppl_from_bram_2;//or if b is active, store that
    from b
238.
239.                            game_state <= game_state+1;
240.                        end
241.                    end else begin
242.                        write_ppl_1 <= 0;
243.                        write_ppl_2 <= 0;
244.
245.                        if (ppl_bram) addr_ppl_1 <= lookup_addr[2]; //ask for upper right from A
246.                        else addr_ppl_2 <= lookup_addr[2];  //ask for upper right from B
247.
248.
249.                        if (ppl_bram) surroundings[0] <= ppl_from_bram_1; //data present from upper
    left lookup. store it
250.                        else surroundings[0] <= ppl_from_bram_2;//or if b is active, store that from b
251.
252.                        game_state <= game_state+1;
253.                     end
254.                end
255.          LOOKUP_LEFT: begin
256.                    if (ppl_bram) addr_ppl_1 <= lookup_addr[3]; //ask for direct left from A
257.                    else addr_ppl_2 <= lookup_addr[3];  //ask for direct left from B
258.
259.                    if (ppl_bram) surroundings[1] <= ppl_from_bram_1; //data present from direct above
    lookup. store it
```

```
260.                    else surroundings[1] <= ppl_from_bram_2;//or if b is active, store that from b
261.
262.                    game_state <= game_state+1;
263.                   end
264.           LOOKUP_RIGHT: begin
265.                    if (ppl_bram) addr_ppl_1 <= lookup_addr[4]; //ask for direct right from A
266.                    else addr_ppl_2 <= lookup_addr[4];  //ask for direct right from B
267.
268.                    if (ppl_bram) surroundings[2] <= ppl_from_bram_1; //data present from upper right
    lookup. store it
269.                    else surroundings[2] <= ppl_from_bram_2;//or if b is active, store that from b
270.
271.                    game_state <= game_state+1;
272.                   end
273.           LOOKUP_DOWNLEFT: begin
274.                    if (ppl_bram) addr_ppl_1 <= lookup_addr[5]; //ask for lower left from A
275.                    else addr_ppl_2 <= lookup_addr[5];  //ask for lower left from B
276.
277.                    if (ppl_bram) surroundings[3] <= ppl_from_bram_1; //data present from direct left
    lookup. store it
278.                    else surroundings[3] <= ppl_from_bram_2;//or if b is active, store that from b
279.
280.                    game_state <= game_state+1;
281.                   end
282.           LOOKUP_DOWN: begin
283.                    if (ppl_bram) addr_ppl_1 <= lookup_addr[6]; //ask for direct below from A
284.                    else addr_ppl_2 <= lookup_addr[6];  //ask for direct below from B
285.
286.                    if (ppl_bram) surroundings[4] <= ppl_from_bram_1; //data present from direct right
    lookup. store it
287.                    else surroundings[4] <= ppl_from_bram_2;//or if b is active, store that from b
288.
289.                    game_state <= game_state+1;
290.                   end
291.           LOOKUP_DOWNRIGHT: begin
292.                    if (ppl_bram) addr_ppl_1 <= lookup_addr[7]; //ask for lower right from A
293.                    else addr_ppl_2 <= lookup_addr[7];  //ask for lower right from B
294.
295.                    if (ppl_bram) surroundings[5] <= ppl_from_bram_1; //data present from lower left
    lookup. store it
296.                    else surroundings[5] <= ppl_from_bram_2;//or if b is active, store that from b
297.
298.                    game_state <= game_state+1;
299.                   end
300.           STORE_DOWN: begin
301.                    if (ppl_bram) surroundings[6] <= ppl_from_bram_1; //data present from direct below
    lookup. store it
302.                    else surroundings[6] <= ppl_from_bram_2;//or if b is active, store that from b
303.
304.                    game_state <= game_state+1;
305.                   end
306.           STORE_DOWNRIGHT: begin
307.                    if (ppl_bram) surroundings[6] <= ppl_from_bram_1; //data present from direct below
    lookup. store it
308.                    else surroundings[6] <= ppl_from_bram_2;//or if b is active, store that from b
309.
310.                    if (ppl_bram) surroundings[7] <= ppl_from_bram_1; //data present from lower right
    lookup. store it
311.                    else surroundings[7] <= ppl_from_bram_2;//or if b is active, store that from b
312.
313.                    game_state <= game_state+1;
314.                   end
315.           MOVE_PPL:begin //done retrieving all information
316.
317.                    //how do we check if a bldg exists at edge of screen?
318.                    //in this step we can check whether to generate a bldg
319.                    //check for collisions?
320.
321.                    case (dir)
322. //                        UP: begin //check if person is at surroundings[1]
323. //                              if (surroundings[1][14:8] > 0) begin
324. //                                if (~ppl_bram) addr_ppl_1 <= ind - GRID_WIDTH - 1;
325. //                                else addr_ppl_2 <= ind - GRID_WIDTH - 1;
326. //                                ppl_to_bram <= {food_indicator, food_storage, food_count, age, 5'b0,
    DOWN};
327. //                               end else begin
328. //                                if (~ppl_bram) addr_ppl_1 <= ind - GRID_WIDTH - GRID_WIDTH - 1;
329. //                                else addr_ppl_2 <= ind - GRID_WIDTH - GRID_WIDTH - 1;
330. //                                //ppl_to_bram <= {food_indicator, food_storage, food_count, age,
    5'b0, UP};
331. //                                ppl_to_bram <= {surroundings[1][29:22], food_count, age, 5'b0, UP};
```

```
332. //                              end
333. //                            end
334. //                          UP_RIGHT: begin //check if person is at surroundings[2]
335. //                                      if (surroundings[2][14:8] > 0) begin
336. //                                        if (~ppl_bram) addr_ppl_1 <= ind - GRID_WIDTH - 1;
337. //                                        else addr_ppl_2 <= ind - GRID_WIDTH - 1;
338. //                                        ppl_to_bram <= {food_indicator, food_storage, food_count,
     age, 5'b0, RIGHT};
339. //                                      end else begin
340. //                                        if (~ppl_bram) addr_ppl_1 <=  ind - GRID_WIDTH + 1 -
     GRID_WIDTH - 1; //we store new addr
341. //                                        else addr_ppl_2 <= ind - GRID_WIDTH + 1 - GRID_WIDTH - 1;
342. //                                        ppl_to_bram <= {surroundings[2][29:22], food_count, age,
     5'b0, UP_RIGHT};
343. //                                      end
344. //                                    end
345.                           RIGHT: begin //check if person is at surroundings[4]
346.                                       if (surroundings[4] > 0) begin
347.                                         if (~ppl_bram) addr_ppl_1 <= random_num[11:0];
348.                                         else addr_ppl_2 <= random_num[11:0];
349.
350.                                         if (age + 1 >= MAX_AGE) ppl_to_bram <= 30'b0;
351.                                         else if (food_zone1) begin
352.                                             ppl_counter <= ppl_counter + 1;
353.                                             ppl_to_bram <= {8'b0, food_count + 1, age + 1,
     random_num[1:0]};
354.                                             food_storage1 <= food_storage1 - 1;
355.                                         end else if (food_zone2) begin
356.                                             ppl_counter <= ppl_counter + 1;
357.                                             ppl_to_bram <= {8'b0, food_count + 1, age + 1,
     random_num[1:0]};
358.                                             food_storage2 <= food_storage2 - 1;
359.                                         end else if (food_count - 1 <= 0) ppl_to_bram <= 30'b0;
360.                                         else begin
361.                                             ppl_counter <= ppl_counter + 1;
362.                                             ppl_to_bram <= {8'b0, food_count - 1, age + 1,
     random_num[1:0]};
363.                                             if (surroundings[4][9:2] > 0) begin
364.                                                 num_collisions <= num_collisions + 1;
365.                                                 stored_addr <= ind - GRID_WIDTH - 1;
366.                                             end
367.                                         end
368.                                       end else begin
369.                                         if (~ppl_bram) addr_ppl_1 <= ind - GRID_WIDTH;//we store new
     addr
370.                                         else addr_ppl_2 <= ind - GRID_WIDTH;
371.
372.                                         if (age + 1 >= MAX_AGE) ppl_to_bram <= 30'b0;
373.                                         else if (food_zone1) begin
374.                                             ppl_counter <= ppl_counter + 1;
375.                                             ppl_to_bram <= {8'b0, food_count + 1, age + 1,
     random_num[1:0]};
376.                                             food_storage1 <= food_storage1 -1;
377.                                         end else if (food_zone2) begin
378.                                             ppl_counter <= ppl_counter + 1;
379.                                             ppl_to_bram <= {8'b0, food_count + 1, age + 1,
     random_num[1:0]};
380.                                             food_storage2 <= food_storage2 - 1;
381.                                         end else if (food_count - 1 <= 0) ppl_to_bram <= 30'b0;
382.                                         else begin ppl_to_bram <= {8'b0, food_count - 1, age + 1,
     random_num[1:0]}; ppl_counter <= ppl_counter + 1; end
383.                                       end
384.                                   end
385. //                           DOWN_RIGHT: begin //check if person is at surroundings[7]
386. //                                       if (surroundings[7][14:8] > 0) begin
387. //                                         if (~ppl_bram) addr_ppl_1 <= ind - GRID_WIDTH - 1;
388. //                                         else addr_ppl_2 <= ind - GRID_WIDTH - 1;
389. //                                         ppl_to_bram <= {food_indicator, food_storage, food_count,
     age, 5'b0, LEFT};
390. //                                       end else begin
391. //                                         if (~ppl_bram) addr_ppl_1 <= ind;//we store new addr
392. //                                         else addr_ppl_2 <= ind;
393. //                                         ppl_to_bram <= {surroundings[7][29:22], food_count, age,
     5'b0, DOWN_RIGHT};
394. //                                       end
395. //                                     end
396.                           DOWN: begin //check if person is at surroundings[6]
397.                                     if (surroundings[6] > 0) begin
398.                                       if (~ppl_bram) addr_ppl_1 <= random_num[11:0];
399.                                       else addr_ppl_2 <= random_num[11:0];
400.
```

```
401.                                    if (age + 1 >= MAX_AGE) ppl_to_bram <= 30'b0;
402.                                    else if (food_zone1) begin
403.                                        ppl_counter <= ppl_counter + 1;
404.                                        ppl_to_bram <= {8'b0, food_count + 1, age + 1,
    random_num[1:0]};
405.                                        food_storage1 <= food_storage1 -1;
406.                                    end else if (food_zone2) begin
407.                                        ppl_counter <= ppl_counter + 1;
408.                                        ppl_to_bram <= {8'b0, food_count + 1, age + 1,
    random_num[1:0]};
409.                                        food_storage2 <= food_storage2 - 1;
410.                                    end else if (food_count - 1 <= 0) ppl_to_bram <= 30'b0;
411.                                    else begin
412.                                        ppl_counter <= ppl_counter + 1;
413.                                        ppl_to_bram <= {8'b0, food_count - 1, age + 1,
    random_num[1:0]};
414.                                        if (surroundings[6][9:2] > 0) begin
415.                                            num_collisions <= num_collisions + 1;
416.                                            stored_addr <= ind - GRID_WIDTH - 1;
417.                                        end
418.                                    end
419.                                end else begin
420.                                    if (~ppl_bram) addr_ppl_1 <= ind -1;//we store new addr
421.                                    else addr_ppl_2 <= ind-1;
422.
423.                                    if (age + 1 >= MAX_AGE) ppl_to_bram <= 30'b0;
424.                                    else if (food_zone1) begin
425.                                        ppl_counter <= ppl_counter + 1;
426.                                        ppl_to_bram <= {8'b0, food_count + 1, age + 1,
    random_num[1:0]};
427.                                        food_storage1 <= food_storage1 -1;
428.                                    end else if (food_zone2) begin
429.                                        ppl_counter <= ppl_counter + 1;
430.                                        ppl_to_bram <= {8'b0, food_count + 1, age + 1,
    random_num[1:0]};
431.                                        food_storage2 <= food_storage2 - 1;
432.                                    end else if (food_count - 1 <= 0) ppl_to_bram <= 30'b0;
433.                                    else begin ppl_to_bram <= {8'b0, food_count - 1, age + 1,
    random_num[1:0]};ppl_counter <= ppl_counter + 1; end
434.                                end
435.                            end
436. //                        DOWN_LEFT: begin //check if person is at surroundings[5]
437. //                                    if (surroundings[5][14:8] > 0) begin
438. //                                      if (~ppl_bram) addr_ppl_1 <= ind - GRID_WIDTH - 1;
439. //                                      else addr_ppl_2 <= ind - GRID_WIDTH - 1;
440. //                                      ppl_to_bram <= {food_indicator, food_storage, food_count,
    age, 5'b0, DOWN_LEFT};
441. //                                    end else begin
442. //                                      if (~ppl_bram) addr_ppl_1 <= ind - 2; //we store new addr
443. //                                      else addr_ppl_2 <= ind - 2;
444. //                                      ppl_to_bram <= {surroundings[5][29:22], food_count, age,
    5'b0, DOWN_LEFT};
445. //                                    end
446. //                                 end
447.                            LEFT: begin //check if person is at surroundings[3]
448.                                    if (surroundings[3] > 0) begin
449.                                      if (~ppl_bram) addr_ppl_1 <= random_num[11:0];
450.                                      else addr_ppl_2 <= random_num[11:0];
451.
452.                                      if (age + 1 >= MAX_AGE) ppl_to_bram <= 30'b0;
453.                                      else if (food_zone1) begin
454.                                          ppl_counter <= ppl_counter + 1;
455.                                          ppl_to_bram <= {8'b0, food_count + 1, age + 1,
    random_num[1:0]};
456.                                          food_storage1 <= food_storage1 -1;
457.                                      end else if (food_zone2) begin
458.                                          ppl_counter <= ppl_counter + 1;
459.                                          ppl_to_bram <= {8'b0, food_count + 1, age + 1,
    random_num[1:0]};
460.                                          food_storage2 <= food_storage2 - 1;
461.                                      end else if (food_count - 1 <= 0) ppl_to_bram <= 30'b0;
462.                                      else begin
463.                                          ppl_counter <= ppl_counter + 1;
464.                                          ppl_to_bram <= {8'b0, food_count - 1, age + 1,
    random_num[1:0]};
465.                                          if (surroundings[3][9:2] > 0) begin
466.                                              num_collisions <= num_collisions + 1;
467.                                              stored_addr <= ind - GRID_WIDTH - 1;
468.                                          end
469.                                       end
470.                                    end else begin
```

```
                                    if (~ppl_bram) addr_ppl_1 <= ind - GRID_WIDTH - 2;//we store new addr
                                    else addr_ppl_2 <= ind - GRID_WIDTH - 2;

                                    if (age + 1 >= MAX_AGE) ppl_to_bram <= 30'b0;
                                    else if (food_zone1) begin
                                        ppl_counter <= ppl_counter + 1;
                                        ppl_to_bram <= {8'b0, food_count + 1, age + 1, random_num[1:0]};
                                        food_storage1 <= food_storage1 -1;
                                    end else if (food_zone2) begin
                                        ppl_counter <= ppl_counter + 1;
                                        ppl_to_bram <= {8'b0, food_count + 1, age + 1, random_num[1:0]};
                                        food_storage2 <= food_storage2 - 1;
                                    end else if (food_count - 1 <= 0) ppl_to_bram <= 30'b0;
                                    else begin ppl_to_bram <= {8'b0, food_count - 1, age + 1, random_num[1:0]};ppl_counter <= ppl_counter + 1; end
                                end
                            end
//                          UP_LEFT: begin //check if person is at surroundings[0]
//                                if (surroundings[0][14:8] > 0) begin
//                                  if (~ppl_bram) addr_ppl_1 <= ind - GRID_WIDTH - 1;
//                                  else addr_ppl_2 <= ind - GRID_WIDTH - 1;
//                                  ppl_to_bram <= {food_indicator, food_storage, food_count, age, 5'b0, UP};
//                                end else begin
//                                  if (~ppl_bram) addr_ppl_1 <= ind - GRID_WIDTH - 1 - GRID_WIDTH - 1;//we store new addr
//                                  else addr_ppl_2 <= ind - GRID_WIDTH - 1 - GRID_WIDTH - 1;
//                                  ppl_to_bram <= {surroundings[0][29:22], food_count, age, 5'b0, UP_LEFT};
//                                end
//                            end
                        default : begin
                                    if (~ppl_bram) addr_ppl_1 <= ind - GRID_WIDTH - 1;
                                    else addr_ppl_2 <= ind - GRID_WIDTH - 1;

                                    if (age + 1 >= MAX_AGE) ppl_to_bram <= 30'b0;
                                    else if (food_zone1) begin
                                        ppl_counter <= ppl_counter + 1;
                                        ppl_to_bram <= {8'b0, food_count + 1, age + 1, random_num[1:0]};
                                        food_storage1 <= food_storage1 -1;
                                    end else if (food_zone2) begin
                                        ppl_counter <= ppl_counter + 1;
                                        ppl_to_bram <= {8'b0, food_count + 1, age + 1, random_num[1:0]};
                                        food_storage2 <= food_storage2 - 1;
                                    end else if (food_count - 1 <= 0) ppl_to_bram <= 30'b0;
                                    else begin ppl_to_bram <= {8'b0, food_count - 1, age + 1, random_num[1:0]}; ppl_counter <= ppl_counter + 1; end
                                end
                    endcase

                    if (~ppl_bram) write_ppl_1 <= 1;
                    else write_ppl_2 <= 1;
                    game_state <= SPAWN_PPL;
                end
          SPAWN_PPL:begin
                    if (num_collisions == 0) time_since_last <= time_since_last + 1;
                    else if (num_collisions > 0) begin
                        ppl_counter <= ppl_counter + 1;
                        num_collisions <= num_collisions - 1;
                        if (~ppl_bram) begin write_ppl_1 <= 1; addr_ppl_1 <= stored_addr; end
                        else begin write_ppl_2 <= 1; addr_ppl_2 <= stored_addr; end
                        ppl_to_bram <= {initial_ppl_state, random_num[1:0]};
                    end else if (time_since_last >= 3) begin
                        ppl_counter <= ppl_counter + 1;
                        if (~ppl_bram) begin write_ppl_1 <= 1; addr_ppl_1 <= random_num[11:0]; end
                        else begin write_ppl_2 <= 1; addr_ppl_2 <= random_num[11:0]; end
                        ppl_to_bram <= {initial_ppl_state, random_num[1:0]};
                        time_since_last <= time_since_last - 1;
                        game_state <= (time_since_last == 0) ? FINAL_STATE : SPAWN_PPL;
                    end

                    game_state <= FINAL_STATE;
                end
          FINAL_STATE:begin //final state (if you want...or you have 1110 and 1111 if you need them
                    write_ppl_1 <= 0;
                    write_ppl_2 <= 0;
```

```
542.                    if (repl_food == 5'd31) begin
543.                        food_storage1 <= food_storage1 + 31;
544.                        food_storage2 <= food_storage2 + 31;
545.                        repl_food <= 0;
546.                    end
547.
548.                    if (ind >= PPL_BRAM_SIZE) begin
549.                        if (game_step) begin
550.                            build <= (ppl_counter%16 > 0) ? 1 : 0;
551.                            repl_food <= repl_food + 1;
552.                            game_state <= CLEAR_BRAM;
553.                            ppl_bram <= ~ppl_bram;
554.                            ind <= 0; //increment count (cell for transer)
555.                        end
556.                    end else begin
557.                        game_state <= LOOKUP_CURR;
558.                        ind <= ind + 1;
559.                    end
560.                end
561.          CLEAR_BRAM: begin
562.                       if (ind <= PPL_BRAM_SIZE) begin
563.                            ind <= ind + 1;
564.                            if (ppl_bram) begin //writing to 2
565.                                addr_ppl_2 <= ind;
566.                                write_ppl_2 <= 1;
567.                                write_ppl_1 <= 0;
568.                                ppl_to_bram <= 30'b0;
569.                            end else begin //writing to 1
570.                                addr_ppl_1 <= ind;
571.                                write_ppl_1 <= 1;
572.                                write_ppl_2 <= 0;
573.                                ppl_to_bram <= 30'b0;
574.                            end
575.                        end else begin
576.                            game_state <= (init) ? INITIALIZE : LOOKUP_CURR;
577.                            ppl_counter <= 0;
578.                            ind <= 0;
579.                            write_ppl_1 <= 0;
580.                            write_ppl_2 <= 0;
581.                        end
582.                    end
583.          default : begin //always wear seatbelts
584.                      write_ppl_1 <= 0;
585.                      write_ppl_2 <= 0;
586.                      game_state <= 0;
587.                    end
588.        endcase
589.
590.
591.    if (rst_in) begin //put at the end to override any assignment above
592.        write_ppl_1  <= 0;
593.        write_ppl_2  <= 0;
594.        ppl_bram   <= 0;
595.        ind   <= 0;
596.        init <= 1;
597.        game_state   <= CLEAR_BRAM;
598.        init_counter <= 6'd50;
599.        time_since_last <= 0;
600.        num_collisions <= 0;
601.        food_storage1 <= 10'd256;
602.        food_storage2 <= 10'd256;
603.        repl_food <= 0;
604.        ppl_counter <= 0;
605.        init_food <= 10'd127;
606.      end
607.    end
608.
609. endmodule
```

Modulus

```
1.  module modulus(input [12:0] ind, output logic [1:0] mod);
2.      always_comb begin
3.          case (ind)
4.              //ind % GRID_WIDTH (96) == 0
5.              13'd0: mod = 0;
6.              13'd96: mod = 0;
7.              13'd192: mod = 0;
```

```
8.              13'd288: mod = 0;
9.              13'd384: mod = 0;
10.             13'd480: mod = 0;
11.             13'd576: mod = 0;
12.             13'd672: mod = 0;
13.             13'd768: mod = 0;
14.             13'd864: mod = 0;
15.             13'd960: mod = 0;
16.             13'd1056: mod = 0;
17.             13'd1152: mod = 0;
18.             13'd1248: mod = 0;
19.             13'd1344: mod = 0;
20.             13'd1440: mod = 0;
21.             13'd1536: mod = 0;
22.             13'd1632: mod = 0;
23.             13'd1728: mod = 0;
24.             13'd1824: mod = 0;
25.             13'd1920: mod = 0;
26.             13'd2016: mod = 0;
27.             13'd2112: mod = 0;
28.             13'd2208: mod = 0;
29.             13'd2304: mod = 0;
30.             13'd2400: mod = 0;
31.             13'd2496: mod = 0;
32.             13'd2592: mod = 0;
33.             13'd2688: mod = 0;
34.             13'd2784: mod = 0;
35.             13'd2880: mod = 0;
36.             13'd2976: mod = 0;
37.             13'd3072: mod = 0;
38.             13'd3168: mod = 0;
39.             13'd3264: mod = 0;
40.             13'd3360: mod = 0;
41.             13'd3456: mod = 0;
42.             13'd3552: mod = 0;
43.             13'd3648: mod = 0;
44.             13'd3744: mod = 0;
45.             13'd3840: mod = 0;
46.             13'd3936: mod = 0;
47.             13'd4032: mod = 0;
48.             13'd4128: mod = 0;
49.             13'd4224: mod = 0;
50.             13'd4320: mod = 0;
51.             13'd4416: mod = 0;
52.             13'd4512: mod = 0;
53.             13'd4608: mod = 0;
54.             13'd4704: mod = 0;
55.             13'd4800: mod = 0;
56.             13'd4896: mod = 0;
57.             13'd4992: mod = 0;
58.             13'd5088: mod = 0;
59.             13'd5184: mod = 0;
60.             13'd5280: mod = 0;
61.             13'd5376: mod = 0;
62.             13'd5472: mod = 0;
63.             13'd5568: mod = 0;
64.             13'd5664: mod = 0;
65.             13'd5760: mod = 0;
66.             13'd5856: mod = 0;
67.             13'd5952: mod = 0;
68.             13'd6048: mod = 0;
69.             13'd6144: mod = 0;
70.             13'd6240: mod = 0;
71.             13'd6336: mod = 0;
72.             13'd6432: mod = 0;
73.             13'd6528: mod = 0;
74.             13'd6624: mod = 0;
75.             13'd6720: mod = 0;
76.             13'd6816: mod = 0;
77.             //ind % GRID_WIDTH (96) == 95
78.             13'd95: mod = 1;
79.             13'd191: mod = 1;
80.             13'd287: mod = 1;
81.             13'd383: mod = 1;
82.             13'd479: mod = 1;
83.             13'd575: mod = 1;
84.             13'd671: mod = 1;
85.             13'd767: mod = 1;
86.             13'd863: mod = 1;
87.             13'd959: mod = 1;
88.             13'd1055: mod = 1;
```

```
89.             13'd1151: mod = 1;
90.             13'd1247: mod = 1;
91.             13'd1343: mod = 1;
92.             13'd1439: mod = 1;
93.             13'd1535: mod = 1;
94.             13'd1631: mod = 1;
95.             13'd1727: mod = 1;
96.             13'd1823: mod = 1;
97.             13'd1919: mod = 1;
98.             13'd2015: mod = 1;
99.             13'd2111: mod = 1;
100.            13'd2207: mod = 1;
101.            13'd2303: mod = 1;
102.            13'd2399: mod = 1;
103.            13'd2495: mod = 1;
104.            13'd2591: mod = 1;
105.            13'd2687: mod = 1;
106.            13'd2783: mod = 1;
107.            13'd2879: mod = 1;
108.            13'd2975: mod = 1;
109.            13'd3071: mod = 1;
110.            13'd3167: mod = 1;
111.            13'd3263: mod = 1;
112.            13'd3359: mod = 1;
113.            13'd3455: mod = 1;
114.            13'd3551: mod = 1;
115.            13'd3647: mod = 1;
116.            13'd3743: mod = 1;
117.            13'd3839: mod = 1;
118.            13'd3935: mod = 1;
119.            13'd4031: mod = 1;
120.            13'd4127: mod = 1;
121.            13'd4223: mod = 1;
122.            13'd4319: mod = 1;
123.            13'd4415: mod = 1;
124.            13'd4511: mod = 1;
125.            13'd4607: mod = 1;
126.            13'd4703: mod = 1;
127.            13'd4799: mod = 1;
128.            13'd4895: mod = 1;
129.            13'd4991: mod = 1;
130.            13'd5087: mod = 1;
131.            13'd5183: mod = 1;
132.            13'd5279: mod = 1;
133.            13'd5375: mod = 1;
134.            13'd5471: mod = 1;
135.            13'd5567: mod = 1;
136.            13'd5663: mod = 1;
137.            13'd5759: mod = 1;
138.            13'd5855: mod = 1;
139.            13'd5951: mod = 1;
140.            13'd6047: mod = 1;
141.            13'd6143: mod = 1;
142.            13'd6239: mod = 1;
143.            13'd6335: mod = 1;
144.            13'd6431: mod = 1;
145.            13'd6527: mod = 1;
146.            13'd6623: mod = 1;
147.            13'd6719: mod = 1;
148.            13'd6815: mod = 1;
149.            13'd6911: mod = 1;
150.            default: mod = 2;
151.        endcase
152.    end
153.endmodule
```

Drawing

```
1.  module drawing(
2.     input clk_in, //65MHz clock
3.     input rst_in,
4.     input hsync, vsync, blank,
5.     input [10:0] hcount,
6.     input [9:0] vcount,
7.     input [29:0] bram_data, //extracted from the people BRAM
8.     input flood,
9.     input game_step,
10.    input build,
```

```
    output logic [12:0] address, //BRAM address to look up in people BRAM
    output logic [3:0] vga_r,
    output logic [3:0] vga_b,
    output logic [3:0] vga_g,
    output logic vga_hs,
    output logic vga_vs
     );

    localparam ISLAND_WIDTH = 96; //in grid spaces
    localparam ISLAND_HEIGHT = 72; //in grid spaces
    localparam LEFT_EDGE = 128;
    localparam RIGHT_EDGE = 896;
    localparam TOP_EDGE = 96;
    localparam BOTTOM_EDGE = 672;

     logic [11:0] rgb;
     logic [10:0] grid_x;
     logic [9:0] grid_y;
     assign grid_x = (hcount-LEFT_EDGE) >> 3;
     assign grid_y = (vcount-TOP_EDGE) >> 3;
     //assign address = ISLAND_WIDTH*grid_y + grid_x;

     logic [11:0] pixel;  // game's pixel  // r=11:8, g=7:4, b=3:0
     logic [11:0] ppl_pixel;

     //PEOPLE AND FOOD
     people_food (.hcount_in(hcount),.vcount_in(vcount),.data(bram_data),.pixel_out(ppl_pixel));

     //BUILDING
     logic [11:0] bldg_pixel;
     building (.hcount_in(hcount), .vcount_in(vcount), .clk_in(clk_in), .rst_in(rst_in),
    .game_step(build), .flood(flood), .pixel_out(bldg_pixel));

     //WATER
     logic [11:0] water_pixel;
     water (.hcount_in(hcount), .vcount_in(vcount), .flood(flood), .pixel_out(water_pixel));

     assign pixel = water_pixel | ppl_pixel | bldg_pixel;

     logic b,hs,vs;
     always_ff @(posedge clk_in) begin
          hs <= hsync;
          vs <= vsync;
          b <= blank;
          address <= ((hcount >= LEFT_EDGE && hcount < RIGHT_EDGE) &&
             (vcount >= TOP_EDGE && vcount <BOTTOM_EDGE)) ? ISLAND_WIDTH*grid_y + grid_x : 0;
          rgb <= (pixel == 12'b0) ? 12'hFF0 : pixel;
      end

     // the following lines are required for the Nexys4 VGA circuit - do not change
     assign vga_r = ~b ? rgb[11:8]: 0;
     assign vga_g = ~b ? rgb[7:4] : 0;
     assign vga_b = ~b ? rgb[3:0] : 0;

     assign vga_hs = ~hs;
     assign vga_vs = ~vs;

endmodule //drawing
```

People and Food

```
module people_food
    #(parameter WIDTH = 8,            // default width: 8 pixels, 1 grid space
                HEIGHT = 8,           // default height: 8 pixels, 1 grid space
                PEOPLE_COLOR = 12'hF00, // default color: red
                FOOD_COLOR = 12'h0F0)  //defaul color: green
    (input [10:0] hcount_in,
     input [9:0] vcount_in,
     input [29:0] data,
     output logic [11:0] pixel_out);

    //ISLAND PARAMS
    localparam ISLAND_WIDTH = 96; //in grid spaces
    localparam ISLAND_HEIGHT = 72; //in grid spaces
    localparam LEFT_EDGE = 128;
    localparam RIGHT_EDGE = 896;
    localparam TOP_EDGE = 96;
    localparam BOTTOM_EDGE = 672;
```

```
18.
19.     //FOOD ZONE PARAMS
20.     localparam ZONE_WIDTH = 128;
21.     localparam ZONE1_LEFT = 256; //start of left food zone
22.     localparam ZONE1_RIGHT = ZONE1_LEFT + ZONE_WIDTH; //end of left food zone
23.     localparam ZONE2_RIGHT = 768; //end of right food zone
24.     localparam ZONE2_LEFT = ZONE2_RIGHT - ZONE_WIDTH; //start of right food zone
25.     localparam ZONE_TOP = 192; //top of both food zones
26.     localparam ZONE_BOTTOM = 576; //bottom of both food zones
27.
28.     always_comb begin
29.        if (data>30'b0 && (hcount_in >= LEFT_EDGE && hcount_in < RIGHT_EDGE) &&
30.           (vcount_in >= TOP_EDGE && vcount_in <BOTTOM_EDGE))begin // if people information is not empty
    draw a person
31.           pixel_out = PEOPLE_COLOR;
32.        end else if (((hcount_in >= ZONE1_LEFT && hcount_in < ZONE1_RIGHT) ||
33.                     (hcount_in >= ZONE2_LEFT && hcount_in < ZONE2_RIGHT)) &&
34.                     (vcount_in >= ZONE_TOP && vcount_in < ZONE_BOTTOM)) begin  // if in food zone
35.           pixel_out = FOOD_COLOR;
36.        end else begin
37.           pixel_out = 0;
38.        end
39.     end
40. endmodule
```

Building

```
1.  module building
2.     #(parameter WIDTH = 64,            // default width: 64 pixels, 8 grid spaces
3.                 HEIGHT = 96,           // default height: 64 pixels, 8 grid spaces
4.                 COLOR_1 = 12'hA52,
5.                 COLOR_2 = 12'h652)  // default color: brown
6.     (input [10:0] hcount_in,
7.      input [9:0] vcount_in,
8.      input clk_in,
9.      input rst_in,
10.     input game_step,
11.     input flood,
12.     output logic [11:0] pixel_out);
13.
14.    //building bounds
15.    localparam LEFT_EDGE = 128;
16.    localparam UPPER_TOP_EDGE = 0;
17.    localparam UPPER_BOTTOM_EDGE = 96;
18.    localparam LOWER_TOP_EDGE = 672;
19.    localparam LOWER_BOTTOM_EDGE = 768;
20.
21.    logic [23:0] grid;
22.    logic [4:0] grid_x;
23.    logic [4:0] ind;
24.    logic [4:0] n;
25.
26.    logic [4:0] bldg_counter;
27.    logic build;
28.
29.    always_ff @(posedge clk_in) begin
30.         if (rst_in) begin
31.             grid <= 24'b0;
32.             bldg_counter <= 5'b0;
33.         end else if (flood) begin //sets flooded buildings to zero
34.             grid[1:0] = 2'b0;
35.             grid[13:10] = 2'b0;
36.             grid[23:22] = 2'b0;
37.         end else begin
38.             if (game_step) begin
39.                 if (bldg_counter < 5'd31) begin
40.                     bldg_counter <= bldg_counter + 1; //only increment bldg_counter every game cycle
41.                 end else begin
42.                     if (~grid[n]) begin
43.                         grid[n] <= 1;
44.                         n <= 0;
45.                         bldg_counter <= 0;
46.                     end else begin
47.                         n <= n+1;
48.                     end
49.                 end
50.             end
51.         end
```

```
52.    end
53.
54.    always_comb begin
55.         grid_x = (hcount_in-LEFT_EDGE) >> 6;
56.         ind = (vcount_in < UPPER_BOTTOM_EDGE) ? grid_x : grid_x + 12;
57.         if ((ind <= 11 && grid[ind]) && (vcount_in >= UPPER_TOP_EDGE && vcount_in < UPPER_BOTTOM_EDGE)
    && //checks to see if it is within upper bounds
58.             (hcount_in >= (grid_x*WIDTH + LEFT_EDGE) && hcount_in < (grid_x*WIDTH + LEFT_EDGE +
    WIDTH))) begin        //checks for horizontal location
59.             pixel_out = (grid_x%2==0) ? COLOR_1 : COLOR_2;
60.         end else if ((ind <= 23 && grid[ind]) && (vcount_in >= LOWER_TOP_EDGE && vcount_in <
    LOWER_BOTTOM_EDGE) && //checks to see if it is within lower bounds
61.             (hcount_in >= (grid_x*WIDTH + LEFT_EDGE) && hcount_in < (grid_x*WIDTH +LEFT_EDGE + WIDTH)))
    begin                //checks for horizontal location
62.             pixel_out = (grid_x%2==0) ? COLOR_2 : COLOR_1;
63.         end else pixel_out = 0;
64.     end
65.
66. endmodule
```

Water

```
1.  module water
2.      #(parameter WIDTH = 128,        //default width: 128 pixels
3.                  HEIGHT = 768,       //default height: 128 pixels
4.                  COLOR = 12'h00F)    //default color: blue
5.      (input [10:0] hcount_in,
6.       input [9:0] vcount_in,
7.       input flood,                   //push btnc to flood
8.       output logic [11:0] pixel_out);
9.
10.      localparam WATER_RISES = 128;  //pixels
11.      localparam LEFT_WATER = 0;    //x starting pixel
12.      localparam RIGHT_WATER = 896;   //x starting pixel
13.
14.      always_comb begin
15.         if (flood)begin
16.             if ((hcount_in >= LEFT_WATER && hcount_in < (LEFT_WATER+WIDTH+WATER_RISES)) ||
17.                 (hcount_in >= (RIGHT_WATER-WATER_RISES) && hcount_in < (RIGHT_WATER+WIDTH)))begin
18.                 pixel_out = COLOR;
19.             end else begin
20.                 pixel_out = 0;
21.             end
22.         end else begin
23.             if ((hcount_in >= LEFT_WATER && hcount_in < (LEFT_WATER+WIDTH)) ||
24.                 (hcount_in >= RIGHT_WATER && hcount_in < (RIGHT_WATER+WIDTH)))begin
25.                 pixel_out = COLOR;
26.             end else begin
27.                 pixel_out = 0;
28.             end
29.         end
30.      end
31. endmodule
```

Random Number Generator

```
1.  module rng(input rst_in, input clk_in, input [2:0] num_sel, output logic [31:0] shifted_res);
2.      logic [31:0] LFSR_32; //hold random values
3.      logic [30:0] LFSR_31; //hold random values
4.      logic [31:0] lfsr_res;
5.
6.      lfsr32_sel lfsr1(.clk_in(clk_in), .num_sel(num_sel), .lfsr_shift(lfsr_res));
7.
8.      always_ff @(posedge clk_in) begin
9.          if (rst_in) begin
10.             LFSR_32 <= lfsr_res;
11.             LFSR_31 <= 31'h23456789; //seed value
12.         end else begin
13.             // We shift 32-bit LFSR twice before XOR-ing with 31-bit LFSR for increased
    pseudorandomness
14.             LFSR_32 <= {LFSR_32[30:0], LFSR_32[0] ^ LFSR_32[15]};
15.             LFSR_31 <= {LFSR_31[29:0], LFSR_31[0] ^ LFSR_31[8]};
16.             shifted_res <= (LFSR_32 ^ LFSR_31) & 16'hFFFF;
17.         end
18.     end
```

```
19. endmodule
20.
21. module lfsr32_sel(input clk_in, input [2:0] num_sel, output logic [31:0] lfsr_shift);
22.     always_comb begin //might need to change to always_ff
23.         case (num_sel)
24.             3'b000: lfsr_shift <= 32'hB4BCD35C;
25.             3'b001: lfsr_shift <= 32'h7A5BC2E3;
26.             3'b010: lfsr_shift <= 32'hB4BCD35C;
27.             3'b011: lfsr_shift <= 32'h29D1E9EB;
28.             3'b100: lfsr_shift <= 32'd798053920;
29.             3'b101: lfsr_shift <= 32'd478902383;
30.             3'b110: lfsr_shift <= 32'd890275340;
31.             3'b111: lfsr_shift <= 32'd374982713;
32.             default: lfsr_shift <= 32'hB4BCD35C;
33.         endcase
34.     end
35.
36. endmodule
```