Design Procedure

The purpose of this lab is to implement what we have learned through this class by creating a fun/cool project that utilizes the DE1-SoC board and peripherals! We chose to create a classic Pacman game.

We are developing a classic Pacman game using SystemVerilog, a hardware description language, to model the electronic systems on an FPGA (Field-Programmable Gate Array). The game will be displayed on a VGA (Video Graphics Array) screen, and the Pacman character will be controlled using the N8 input controller.

Core Features

- Maze Display: The game will feature a maze layout, similar to the classic Pacman games. This maze will be displayed on the VGA screen.
- Pacman Movement: The Pacman character will be controllable in four directions (up, down, left, right) using the N8 controller.
- Ghosts: There will be a ghost in the maze that moves chasing the Pacman. If Pacman collides with any of the ghosts, the game will end.
- Food: Small food pellets will be placed throughout the maze. Pacman must eat all the food to win the game.

Maze

In creating the maze for the Pacman game, we utilize a given VGA Driver. This driver provides x- and y-coordinates of the current pixel, where the origin is the upper-left corner. The user, in turn, provides the driver with a color to draw at that pixel, which must be valid within two clock cycles of the coordinates changing. Colors are expressed as three 8-bit unsigned integers, each specifying the light intensity in its respective color channel: red, green, or blue.

Since the VGA Driver employs a "scanning" method, rather than the user specifying the color (black or white) of an arbitrary pixel, the driver cycles through one pixel at a time based on the outputs x and y. The user then supplies the RGB color for that pixel. For our maze, we need to provide the specific color, which is blue, to the given coordinates. Thus, as the VGA Driver scans through each pixel, we will supply the blue color for the pixels that form the maze walls.

First, we are using a screen size of 96x72 pixels. The Pacman character will be 3x3 pixels. We scale Pacman to be 3x3 instead of 1x1 because we need to create food items that are smaller than Pacman. Therefore, Pacman cannot be the smallest element. The food items will be 1x1 pixel, positioned in the middle of Pacman's 3x3 grid.

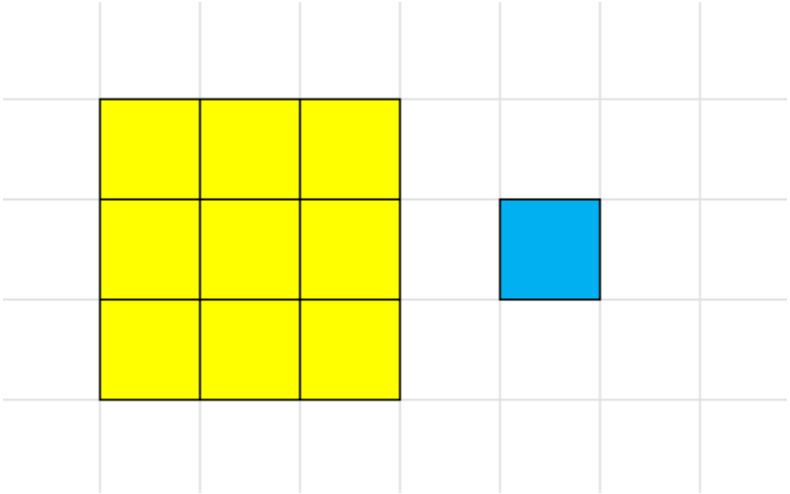


Figure 1. The pixels size for the Pacman and the food

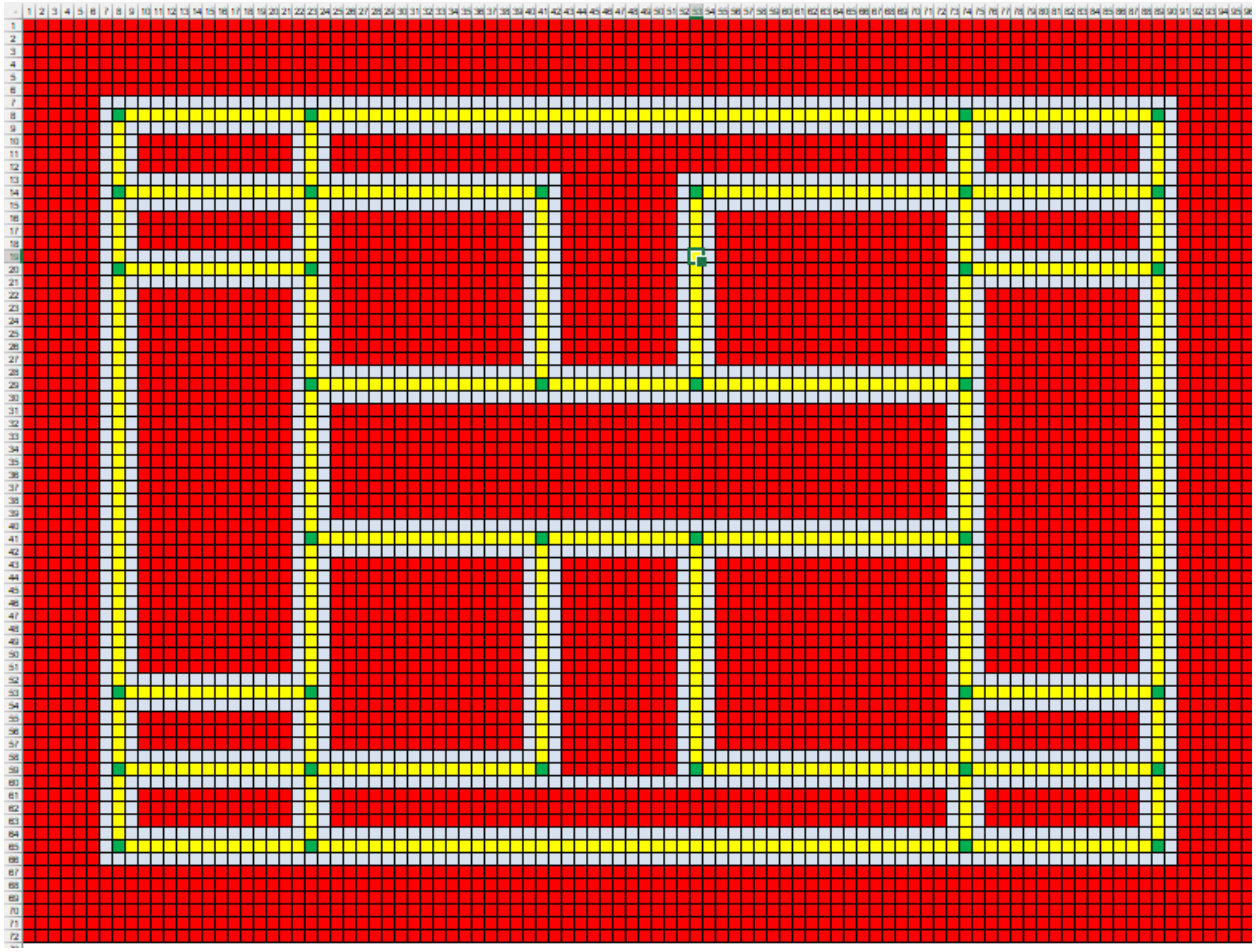


Figure 2. The coordinates for the pacman games.

n the given screenshots, the colors represent various elements of the Pacman game as follows:

- Red represents the maze walls.
- Yellow represents the path movement of Pacman.
- Blue represents the edge tile because Pacman is 3x3 pixels in size.
- Green represents intersections or edge cases.

By using these color codes, we can clearly differentiate between the maze walls, Pacman's path, edge tiles, and intersections. This color scheme helps in visually organizing the game's layout and ensuring that each element is easily identifiable.

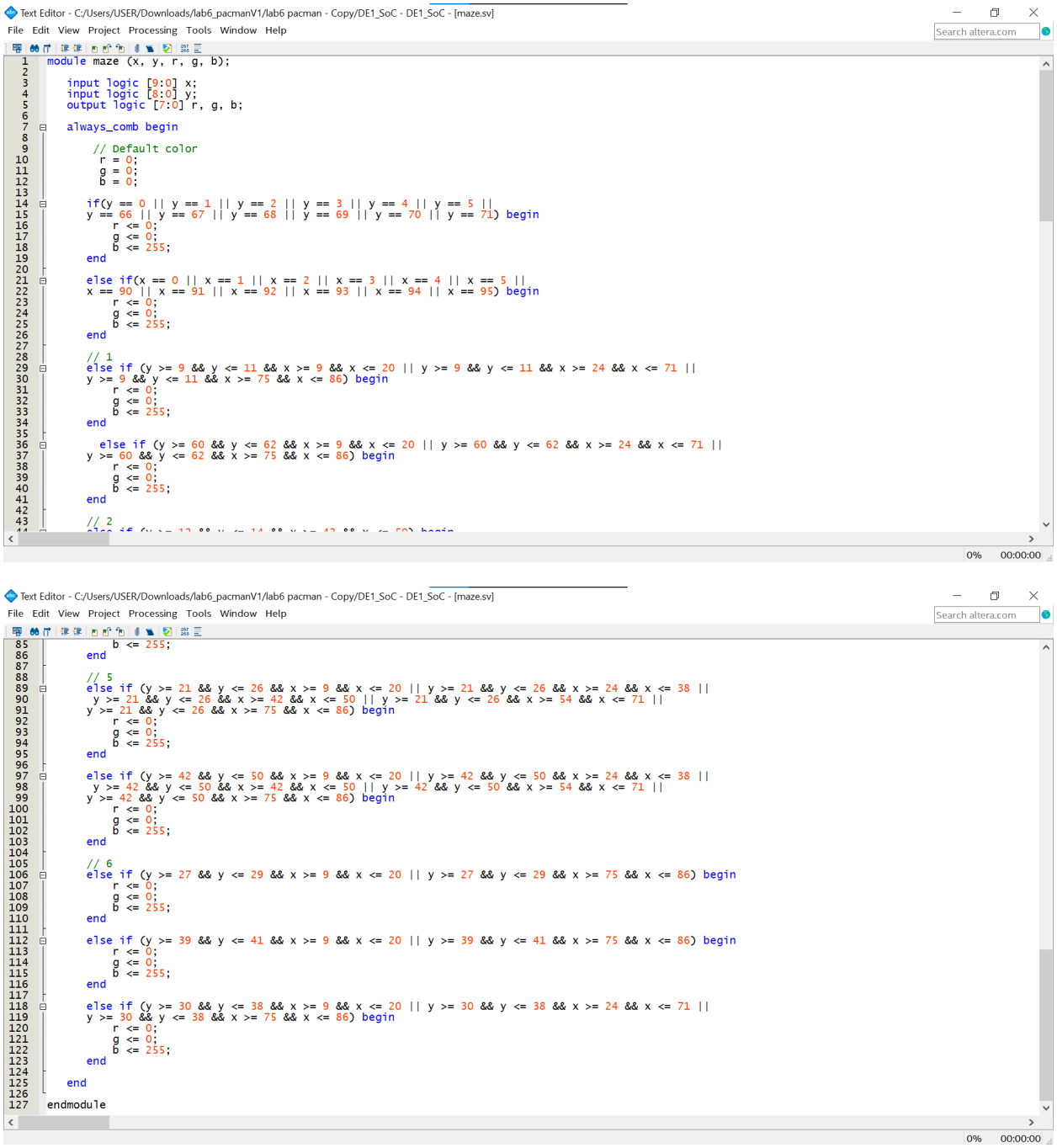


Figure 3. Systemverilog code of maze

Following the figure 2, we can easily hard-code which x and y coordinates that are going to be colored blue to build the maze.

Pacman

For the pacman, we have 5 main modules that are utilized.

The first module is pacman_cont. This module determines the state movement of Pacman, providing the necessary signals to the datapath.

Next is pacman_datapath, which controls the data according to the state determined by pacman_cont. This module manages the actual movement of Pacman within the game.

The pacman_display module is responsible for displaying Pacman at a scale of 3x3 pixels instead of 1x1, ensuring Pacman is visually larger and distinct on the screen.

The user_direction_hold module keeps Pacman moving in the specified direction without the need for the user to repeatedly input the direction. This allows for smoother and more continuous movement based on the user's initial input.

Finally, the pacman module is the top-level module that integrates all the Pacman-related modules, coordinating their functions to create the complete game.

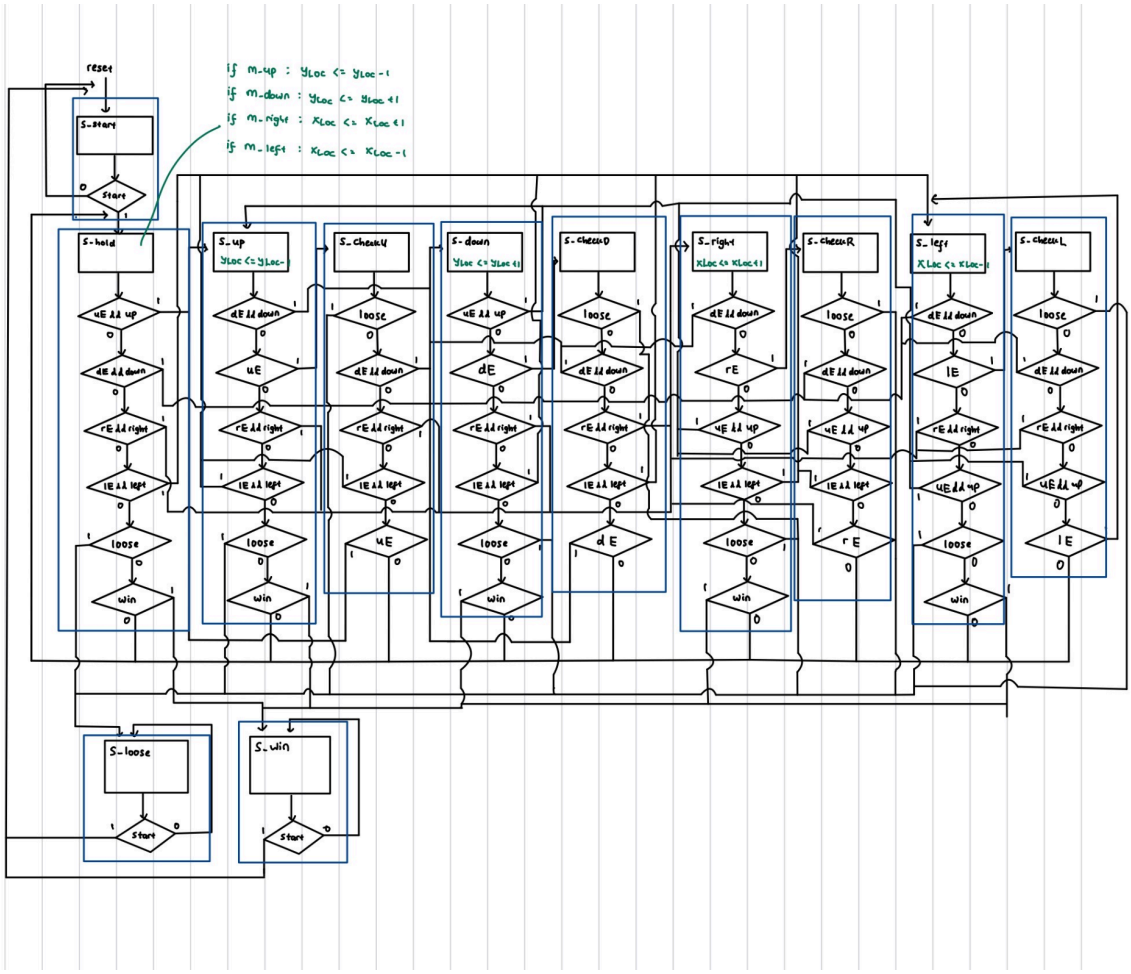


Figure 4. ASMD for Pacman

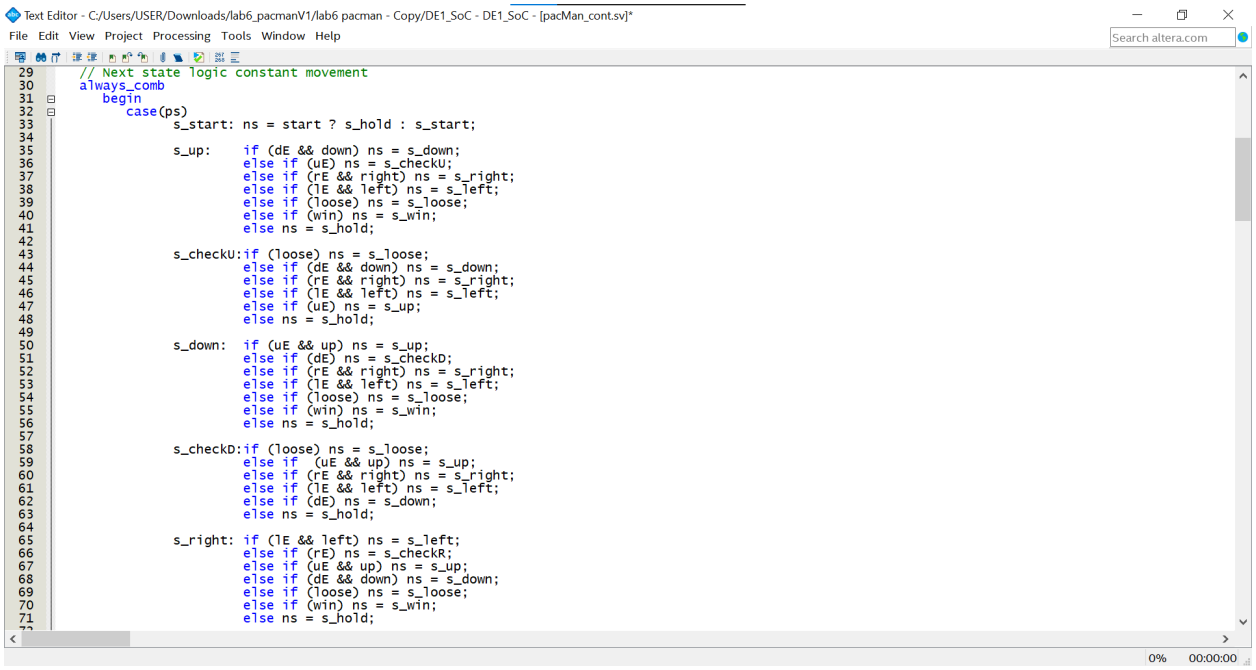
Text Editor - C:/Users/USER/Downloads/lab6_pacmanV1/lab6 pacman - Copy/DE1_SoC - DE1_SoC - [pacMan_cont.sv]*

File Edit View Project Processing Tools Window Help

Search altera.com

```
1  // Pacman control module to control the state of the pacman which include
2  // s_start, s_hold, s_right, s_checkU, s_down, s_checkD,
3  // s_right, s_checkR, s_left, s_checkL, s_win, s_loose
4  module pacMan_cont (
5     input logic clk,
6                 reset,
7                 win, loose,
8                 uE, dE, rE, lE, //maze enable signals
9                 up, down, left, right, start, // controler signals
10    output logic m_up, m_down, m_right, m_left, e_start, m_hold); // output moves
11
12
13    //state variables
14    enum{s_start, s_hold,
15          s_up, s_down, s_right, s_left,
16          s_checkU, s_checkD, s_checkR, s_checkL,
17          s_win, s_loose} ps, ns;
18
19
20    // Controller logic w/synchronous reset
21    always_ff @(posedge clk)
22       if (reset)
23          ps <= s_start;
24       else
25          ps <= ns;
26
27
28
29    // Next state logic constant movement
30    always_comb
31       begin
32          case(ps)
33                s_start: ns = start ? s_hold : s_start;
34
35                s_up:    if (dE && down) ns = s_down;
36                         else if (uE) ns = s_checkU;
37                         else if (rE && right) ns = s_right;
38                         else if (lE && left) ns = s_left;
39                         else if (loose) ns = s_loose;
40                         else if (win) ns = s_win;
41                         else ns = s_hold;
42
43                s_checkU:if (loose) ns = s_loose;
```

0% 00:00:00



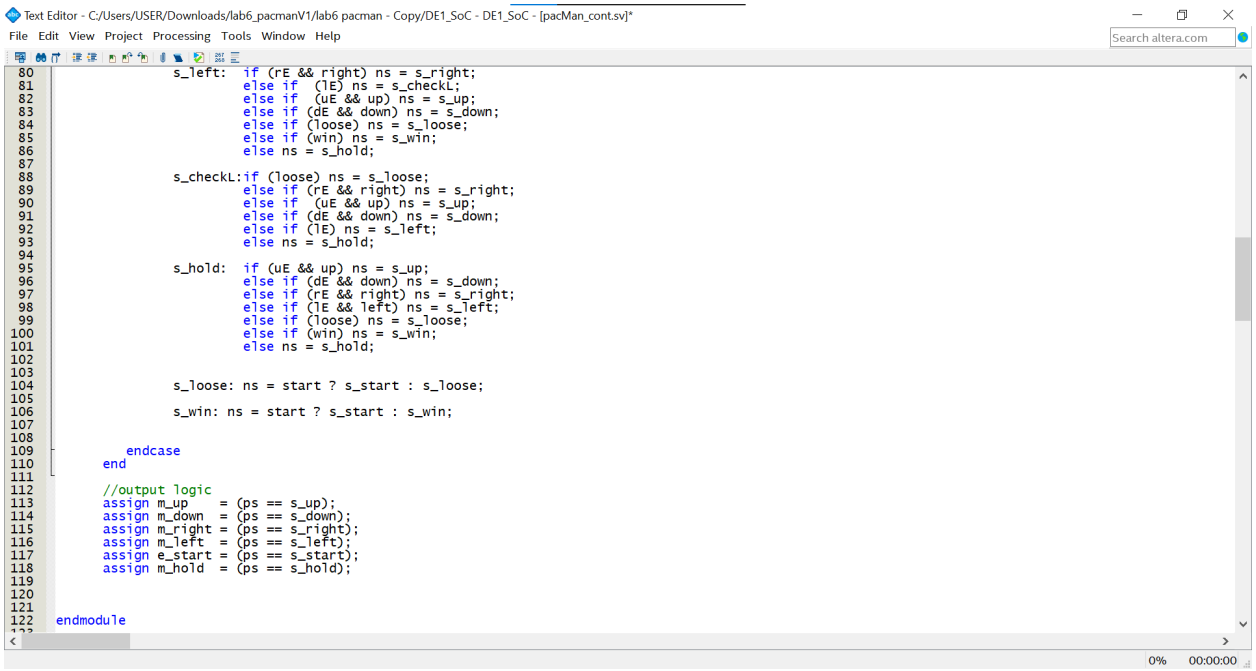


Figure 5. Systemverilog for pacman_cont

Text Editor - C:/Users/USER/Downloads/lab6_pacmanV1/lab6 pacman - Copy/DE1_SoC - DE1_SoC - [pacMan_datapath.sv]*

File Edit View Project Processing Tools Window Help

Search altera.com

```
1  // pacMan_datapath that determines what to do from the signal that is given from pacman_cont
2  // if m_up: yLoc <= yLoc - 1
3  // if m_down: yLoc <= yLoc + 1
4  // if m_right: xLoc <= xLoc + 1
5  // if m_down: xLoc <= xLoc -1
6
7  module pacMan_datapath #(parameter WIDTH = 96, parameter HEIGHT = 72) (
8      input logic clk,
9                  reset,
10                 m_up, m_down, m_right, m_left, e_start, // valid moves from control
11
12     output logic [9:0] xPacLoc,
13     output logic [8:0] yPacLoc);
14
15
16     logic [9:0] xLoc;
17     logic [8:0] yLoc;
18
19
20     always_ff @(posedge clk) begin
21        if (reset || e_start) begin
22           xLoc <= 7;
23           yLoc <= 7;
24           end
25        // limits of the board
26        // ##########################current bug when travleing through board
27        else if (xLoc == WIDTH)
28           xLoc <= 0;
29
30        else if (xLoc == 0)
31           xLoc <= WIDTH;
32
33        else if (yLoc == HEIGHT)
34           yLoc <= 0;
35
36        else if (yLoc == 0)
37           yLoc <= HEIGHT;
38        // Location movment
39        if (m_up)
40           yLoc <= yLoc - 1;
41
42        if(m_down)
43           yLoc <= yLoc + 1;
```

0% 00:00:00

Text Editor - C:/Users/USER/Downloads/lab6_pacmanV1/lab6 pacman - Copy/DE1_SoC - DE1_SoC - [pacMan_datapath.sv]*

File Edit View Project Processing Tools Window Help

Search altera.com

```
17     logic [8:0] yLoc;
18
19
20     always_ff @(posedge clk) begin
21        if (reset || e_start) begin
22           xLoc <= 7;
23           yLoc <= 7;
24           end
25        // limits of the board
26        // ##########################current bug when travleing through board
27        else if (xLoc == WIDTH)
28           xLoc <= 0;
29
30        else if (xLoc == 0)
31           xLoc <= WIDTH;
32
33        else if (yLoc == HEIGHT)
34           yLoc <= 0;
35
36        else if (yLoc == 0)
37           yLoc <= HEIGHT;
38        // Location movment
39        if (m_up)
40           yLoc <= yLoc - 1;
41
42        if(m_down)
43           yLoc <= yLoc + 1;
44
45        if (m_right)
46           xLoc <= xLoc + 1;
47
48        if (m_left)
49           xLoc <= xLoc - 1;
50
51     end // always_ff
52
53
54     //Output logic
55     assign xPacLoc = xLoc;
56     assign yPacLoc = yLoc;
57
58
59  endmodule // pacMan_datapath
```

0% 00:00:00

Figure 6. Systemverilog for pacman_datapath

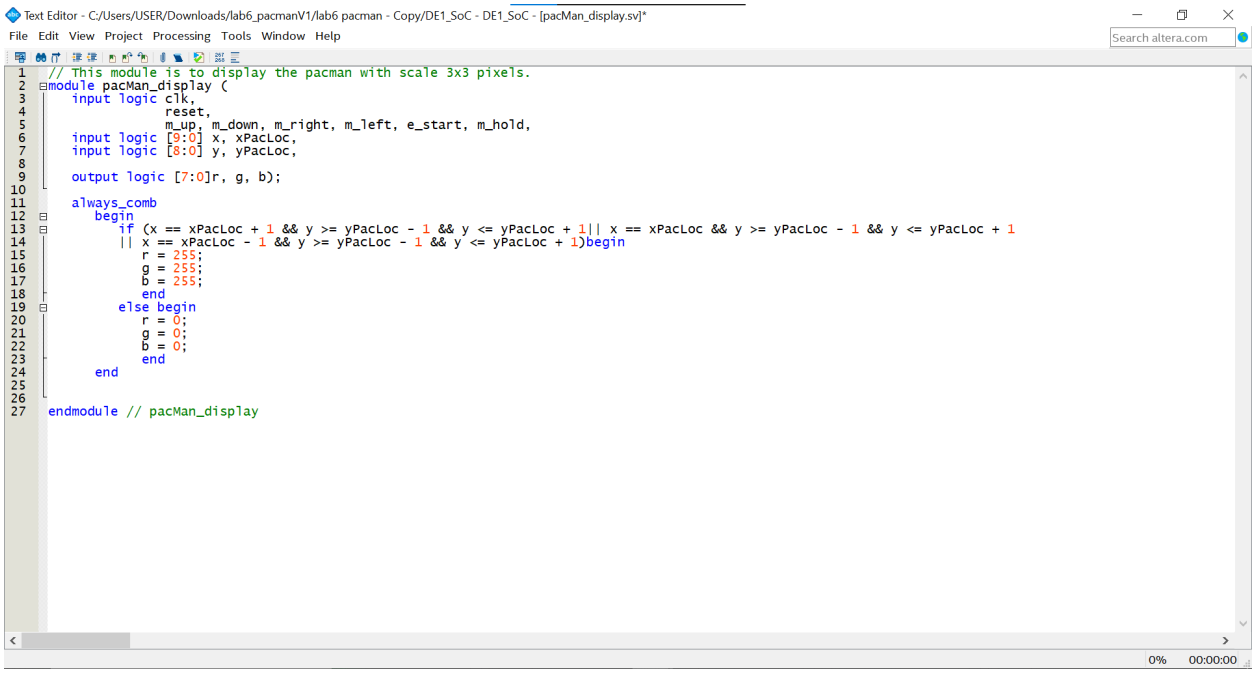


Figure 7. Systemverilog for pacman_display

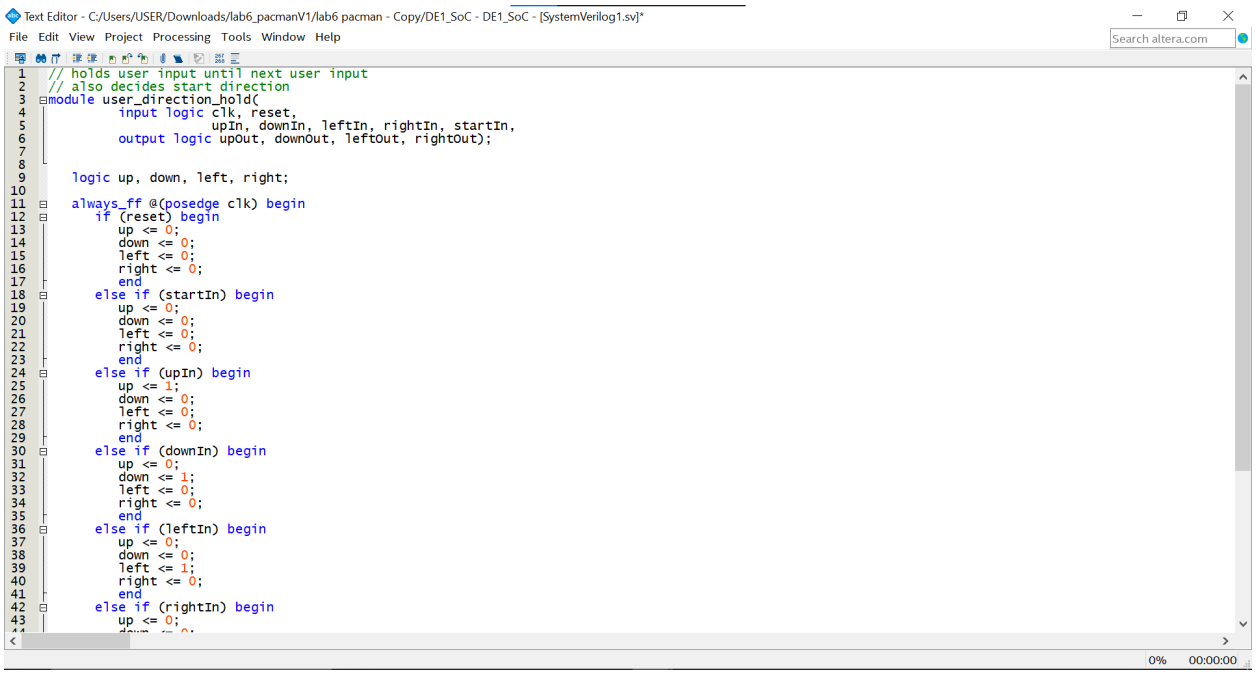
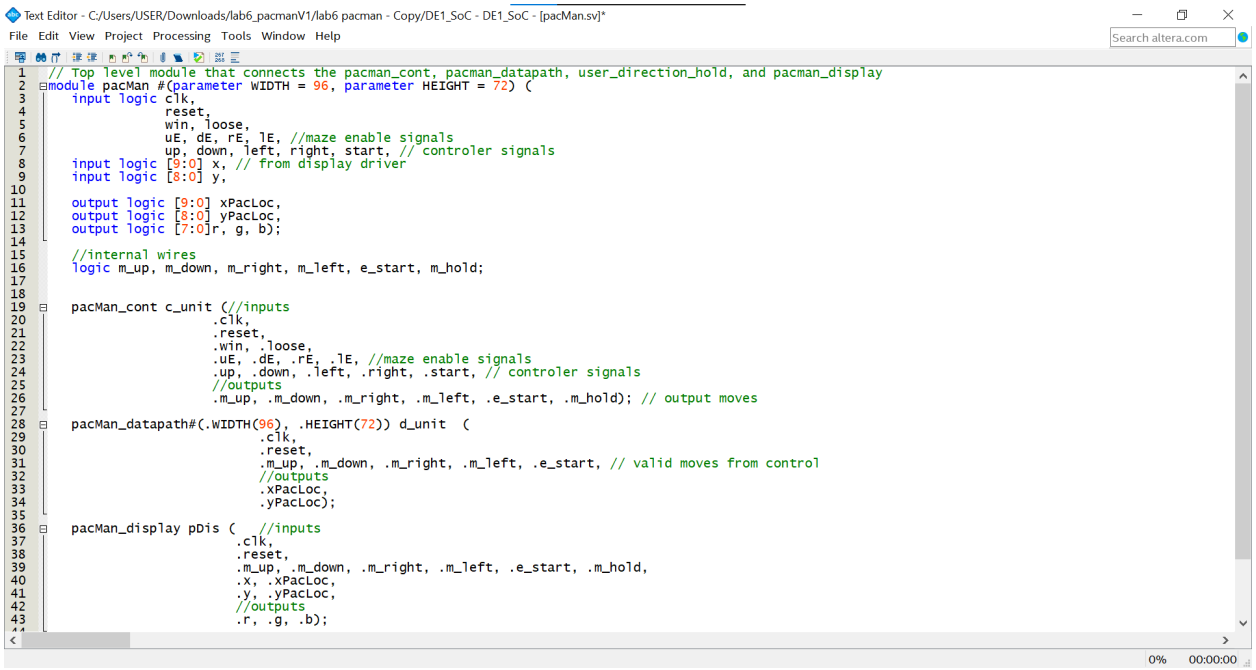


Figure 8. Systemverilog for user_direction_hold



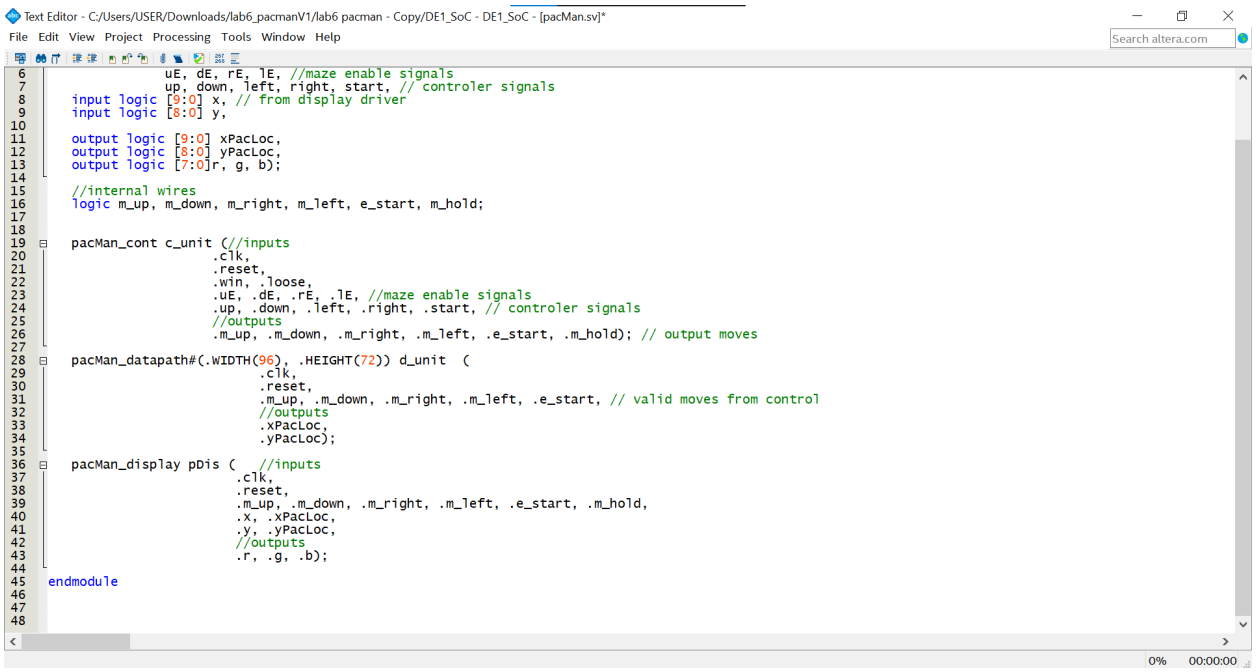


Figure 9. Systemverilog for pacman

In summary, Pacman will move according to the user's input, which can be one of four directions: left, up, right, or down. The user does not need to repeatedly input commands to keep Pacman moving. Once a direction is specified, Pacman will continue moving in that direction until it encounters a wall.

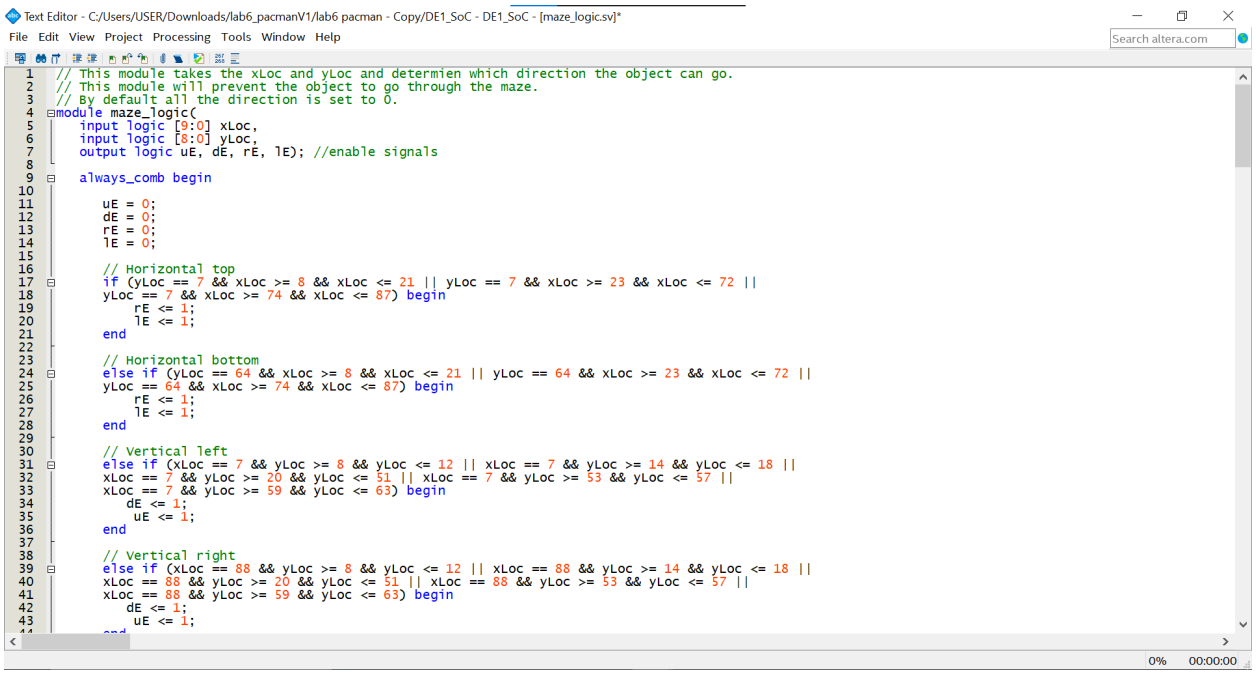
Here's how the movement system works in detail:

- Direction Input: The user inputs a direction (left, up, right, or down) using the control interface.
- Continuous Movement: The user_direction_hold module ensures that Pacman keeps moving in the specified direction continuously. This eliminates the need for the user to repeatedly input the same command to keep Pacman moving.
- State Management: The pacman_cont module manages Pacman's state, determining the direction of movement based on the user's input and the current state of the game.
- Datapath Control: The pacman_datapath module controls Pacman's movement on the screen, updating Pacman's position according to the current state and direction of movement.
- Collision Detection: Pacman will continue to move in the specified direction until it encounters a wall. The game checks for collisions with the maze walls and stops Pacman's movement when a wall is hit.
- Visual Representation: The pacman_display module renders Pacman on the screen, scaled to 3x3 pixels for better visibility. This module ensures that Pacman's movements are accurately displayed according to the direction and position provided by the datapath.
- Top-Level Integration: The pacman module integrates all these functionalities, ensuring that the user's input is processed, Pacman's state is managed, movements are controlled, and the display is updated in a cohesive manner.

This system provides a seamless and intuitive control experience, allowing the user to simply specify the direction and watch Pacman move continuously until an obstacle is encountered.

Maze logic

After, we have implemented the pacman, now we want to create the maze logic because we don't want the pacman to be able to go through the maze. The pacman can only go in the path which is shown in the yellow block in figure 2. To implement this, we hard coded the logic according to the position of the pacman.



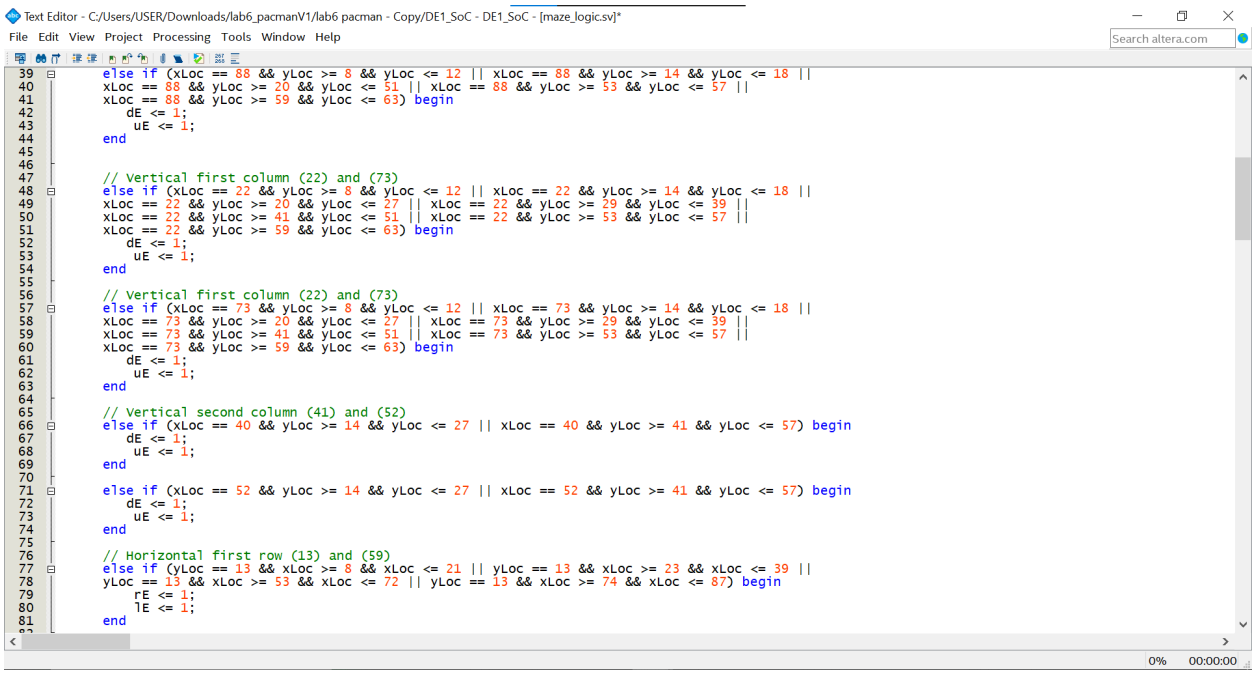


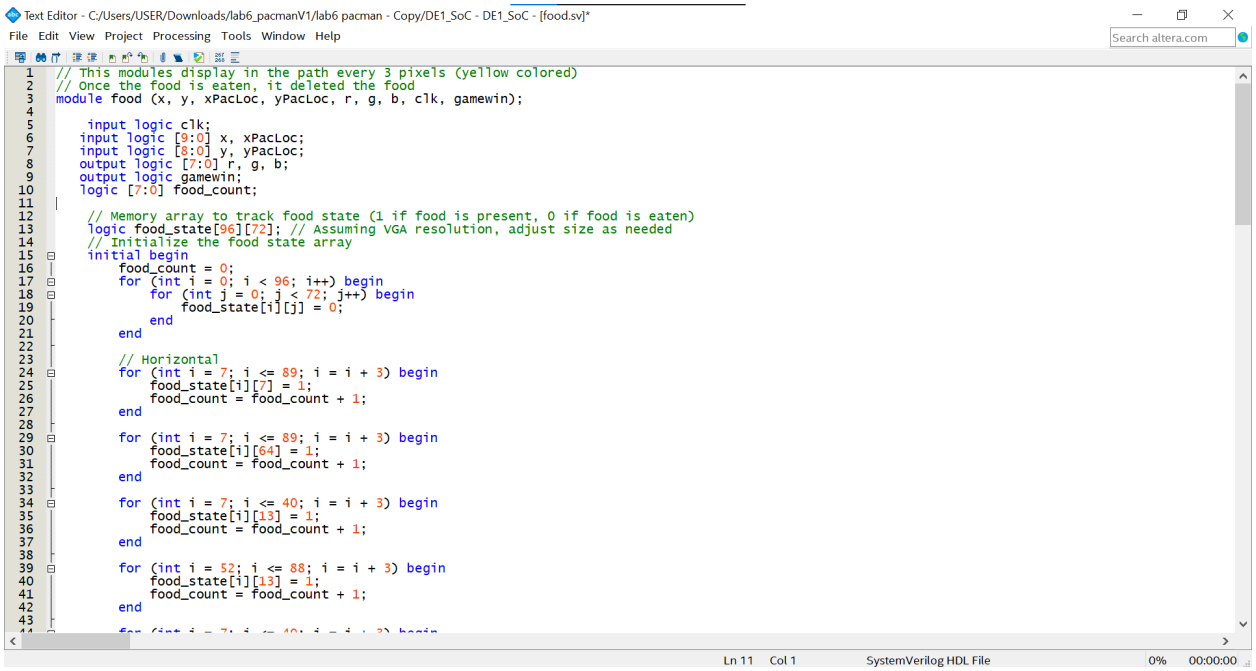
Figure 10. Systemverilog for maze_logic

By implementing this module, Pacman is prevented from moving through the maze walls because any direction leading to a wall will be set to 0.

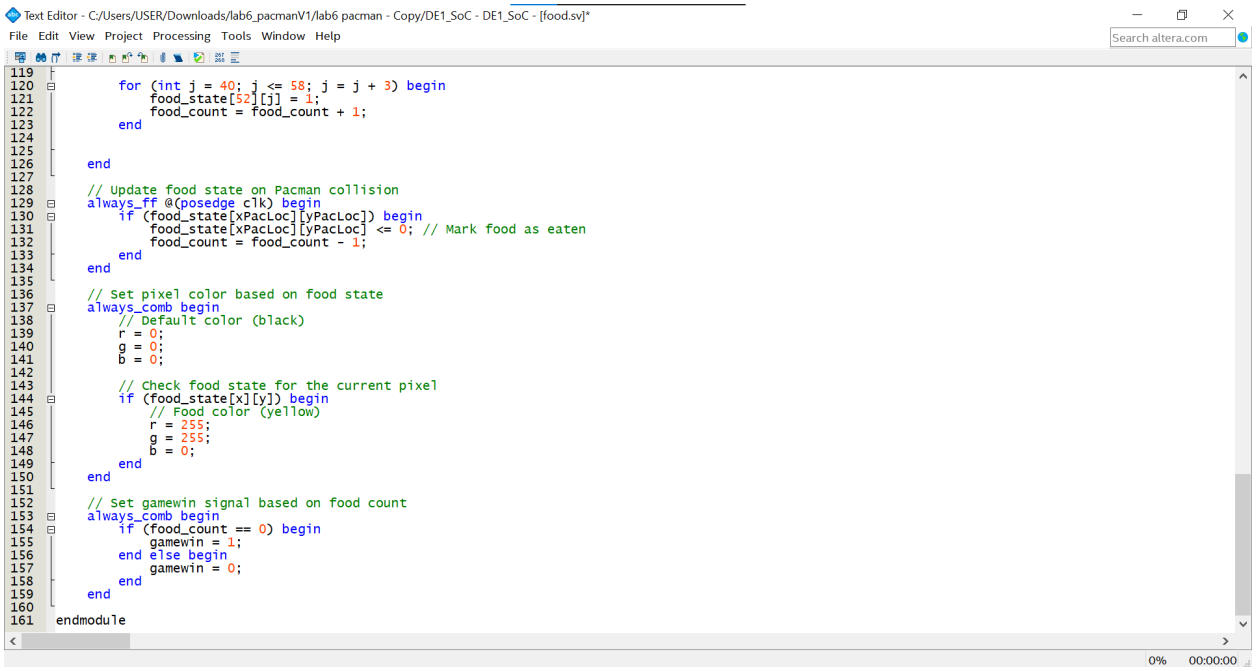
Food

Next, using the same approach we used to build the maze, we will create the food. The food will be placed every three pixels starting from the edge, colored yellow, and sized 1x1. We also need

to implement additional logic so that when Pacman eats the food (when the food location coincides with Pacman's location), it will disappear. We'll use a memory array to track the food state, where 1 indicates food is present and 0 indicates it has been eaten. We assign 1 to every coordinate where we want to place food. When Pacman collides with these coordinates, we assign 0. Finally, we assign the RGB color based on the flag: if it's 1, we assign yellow; otherwise, black. This way, the food will be displayed and will disappear once Pacman eats it.




```
// This modules display in the path every 3 pixels (yellow colored)
// Once the food is eaten, it deleted the food
module food (x, y, xPacLoc, yPacLoc, r, g, b, clk, gamewin);

    input logic clk;
   input logic [9:0] x, xPacLoc;
   input logic [8:0] y, yPacLoc;
   output logic [7:0] r, g, b;
   output logic gamewin;
   logic [7:0] food_count;

    // Memory array to track food state (1 if food is present, 0 if food is eaten)
    logic food_state[96][72]; // Assuming VGA resolution, adjust size as needed
    // Initialize the food state array
    initial begin
        food_count = 0;
        for (int i = 0; i < 96; i++) begin
            for (int j = 0; j < 72; j++) begin
                food_state[i][j] = 0;
            end
        end

        // Horizontal
        for (int i = 7; i <= 89; i = i + 3) begin
            food_state[i][7] = 1;
            food_count = food_count + 1;
        end

        for (int i = 7; i <= 89; i = i + 3) begin
            food_state[i][64] = 1;
            food_count = food_count + 1;
        end

        for (int i = 7; i <= 40; i = i + 3) begin
            food_state[i][13] = 1;
            food_count = food_count + 1;
        end

        for (int i = 52; i <= 88; i = i + 3) begin
            food_state[i][13] = 1;
            food_count = food_count + 1;
        end
```

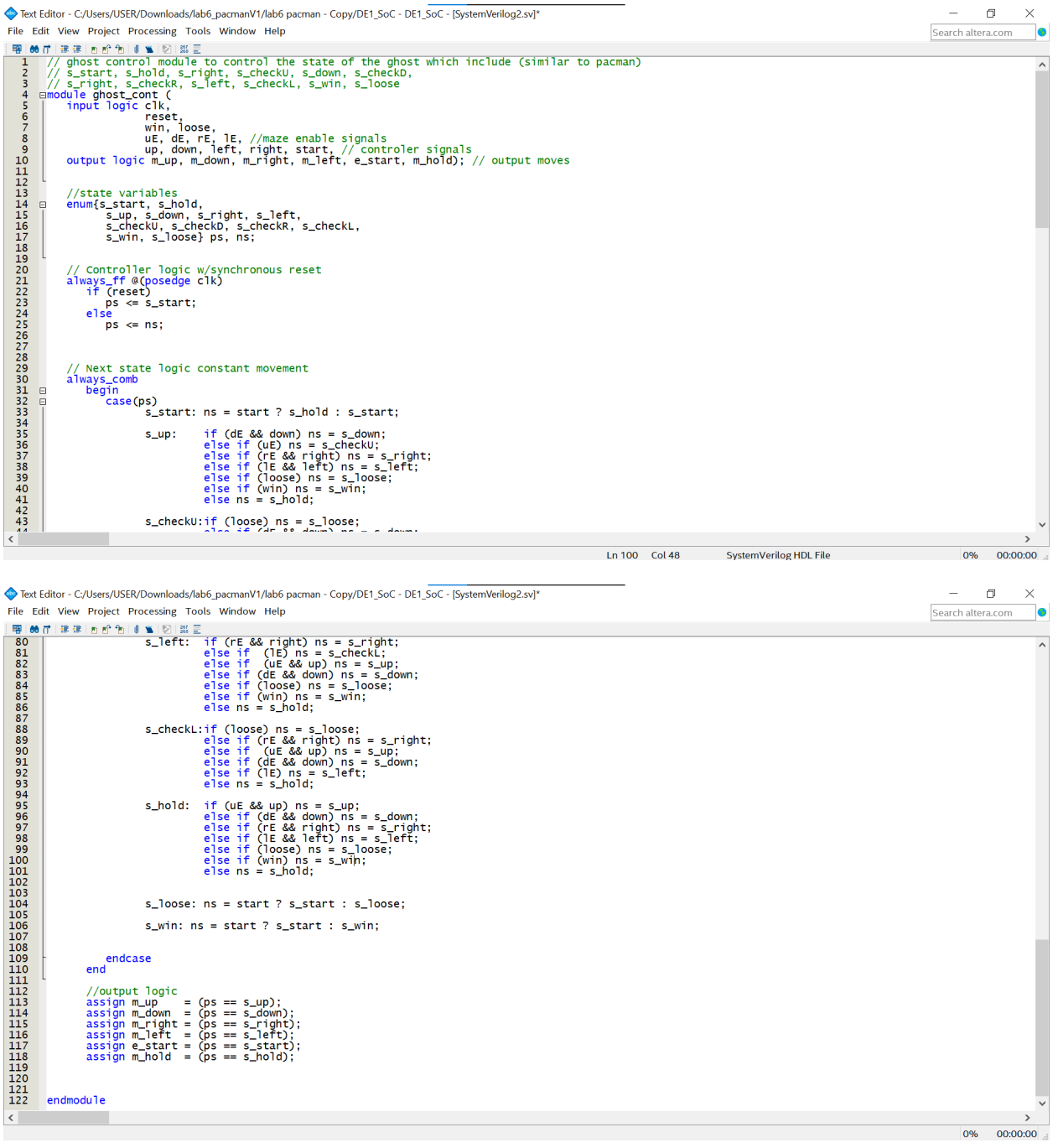



```
        for (int j = 40; j <= 58; j = j + 3) begin
            food_state[52][j] = 1;
            food_count = food_count + 1;
        end


    end

    // Update food state on Pacman collision
    always_ff @(posedge clk) begin
        if (food_state[xPacLoc][yPacLoc]) begin
            food_state[xPacLoc][yPacLoc] <= 0; // Mark food as eaten
            food_count = food_count - 1;
        end
    end

    // Set pixel color based on food state
    always_comb begin
        // Default color (black)
        r = 0;
        g = 0;
        b = 0;

        // Check food state for the current pixel
        if (food_state[x][y]) begin
            // Food color (yellow)
            r = 255;
            g = 255;
            b = 0;
        end
    end

    // Set gamewin signal based on food count
    always_comb begin
        if (food_count == 0) begin
            gamewin = 1;
        end else begin
            gamewin = 0;
        end
    end

endmodule
```

Figure 11. Systemverilog for food

Ghost

Now, for ghosts, the implementation is similar to the pacman. However, instead of having the user input to control the movement, we created the ghost logic module to determine the ghost movement because we want the ghost to chase the pacman.



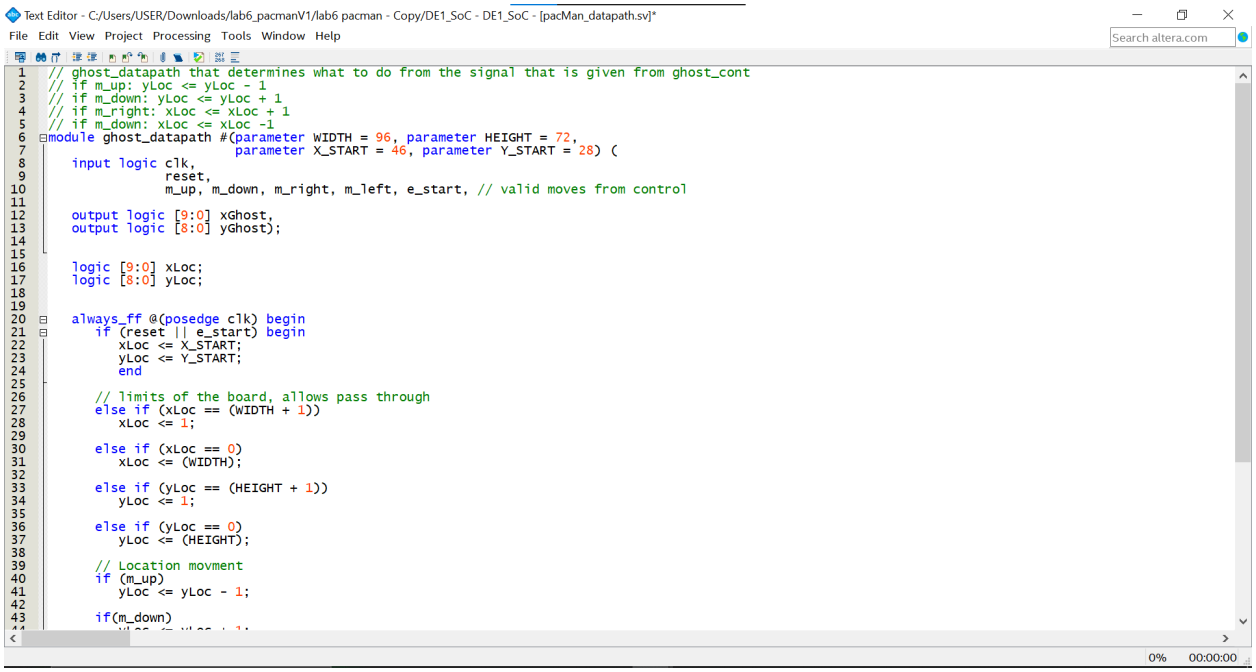
```
// ghost control module to control the state of the ghost which include (similar to pacman)
// s_start, s_hold, s_right, s_checkU, s_down, s_checkD,
// s_right, s_checkR, s_left, s_checkL, s_win, s_loose
module ghost_cont (
    input logic clk,
                reset,
                win, loose,
                uE, dE, rE, lE, //maze enable signals
                up, down, left, right, start, // controler signals
    output logic m_up, m_down, m_right, m_left, e_start, m_hold); // output moves

    //state variables
    enum{s_start, s_hold,
         s_up, s_down, s_right, s_left,
         s_checkU, s_checkD, s_checkR, s_checkL,
         s_win, s_loose} ps, ns;

    // Controller logic w/synchronous reset
    always_ff @(posedge clk)
       if (reset)
          ps <= s_start;
       else
          ps <= ns;


    // Next state logic constant movement
    always_comb
       begin
          case(ps)
                s_start: ns = start ? s_hold : s_start;

                s_up:    if (dE && down) ns = s_down;
                         else if (uE) ns = s_checkU;
                         else if (rE && right) ns = s_right;
                         else if (lE && left) ns = s_left;
                         else if (loose) ns = s_loose;
                         else if (win) ns = s_win;
                         else ns = s_hold;

                s_checkU:if (loose) ns = s_loose;
```

Ln 100 Col 48 SystemVerilog HDL File 0% 00:00:00


```
                s_left:  if (rE && right) ns = s_right;
                         else if  (lE) ns = s_checkL;
                         else if  (uE && up) ns = s_up;
                         else if (dE && down) ns = s_down;
                         else if (loose) ns = s_loose;
                         else if (win) ns = s_win;
                         else ns = s_hold;

                s_checkL:if (loose) ns = s_loose;
                         else if (rE && right) ns = s_right;
                         else if  (uE && up) ns = s_up;
                         else if (dE && down) ns = s_down;
                         else if (lE) ns = s_left;
                         else ns = s_hold;

                s_hold:  if (uE && up) ns = s_up;
                         else if (dE && down) ns = s_down;
                         else if (rE && right) ns = s_right;
                         else if (lE && left) ns = s_left;
                         else if (loose) ns = s_loose;
                         else if (win) ns = s_win;
                         else ns = s_hold;


                s_loose: ns = start ? s_start : s_loose;

                s_win: ns = start ? s_start : s_win;


          endcase
    end

    //output logic
    assign m_up    = (ps == s_up);
    assign m_down  = (ps == s_down);
    assign m_right = (ps == s_right);
    assign m_left  = (ps == s_left);
    assign e_start = (ps == s_start);
    assign m_hold  = (ps == s_hold);


endmodule
```

0% 00:00:00

Figure 12. Systemverilog for ghost_cont



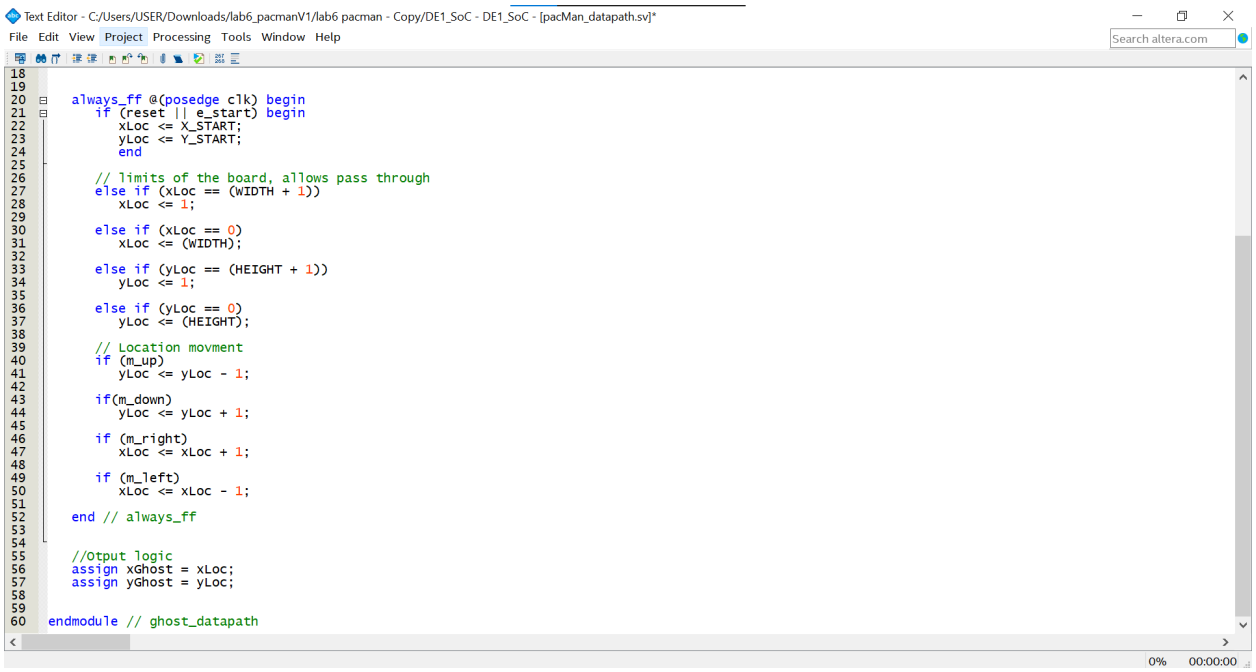


Figure 13. Systemverilog for ghost_datapath

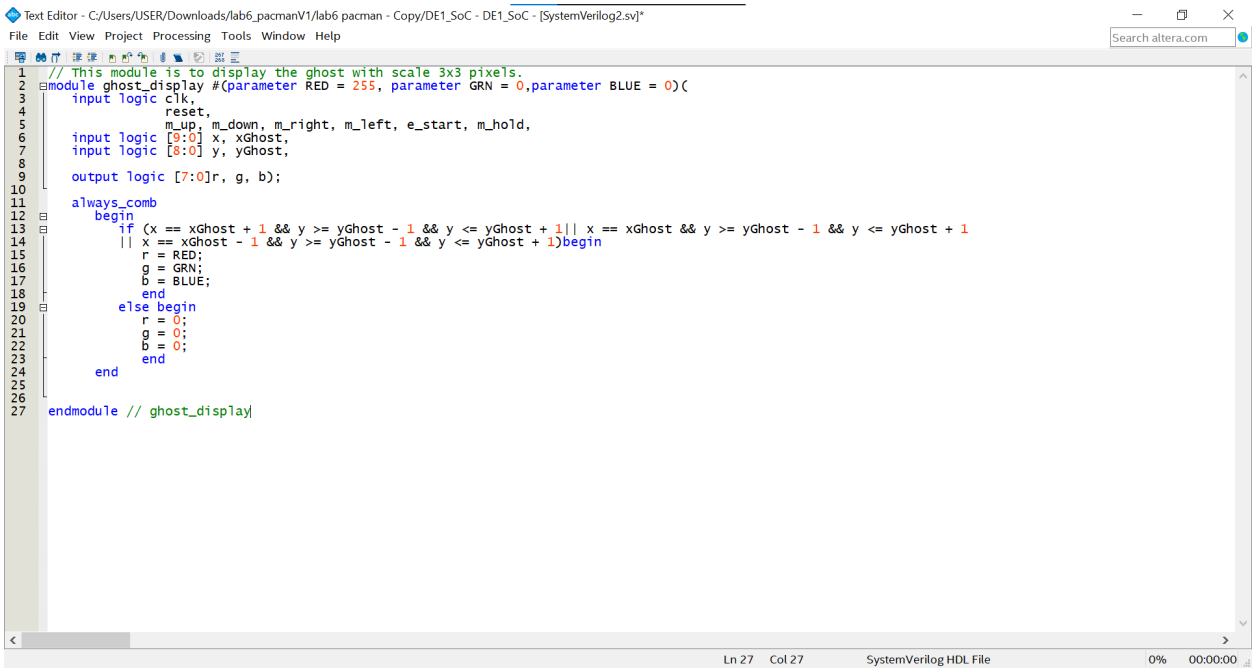


Figure 14. Systemverilog for ghost_display

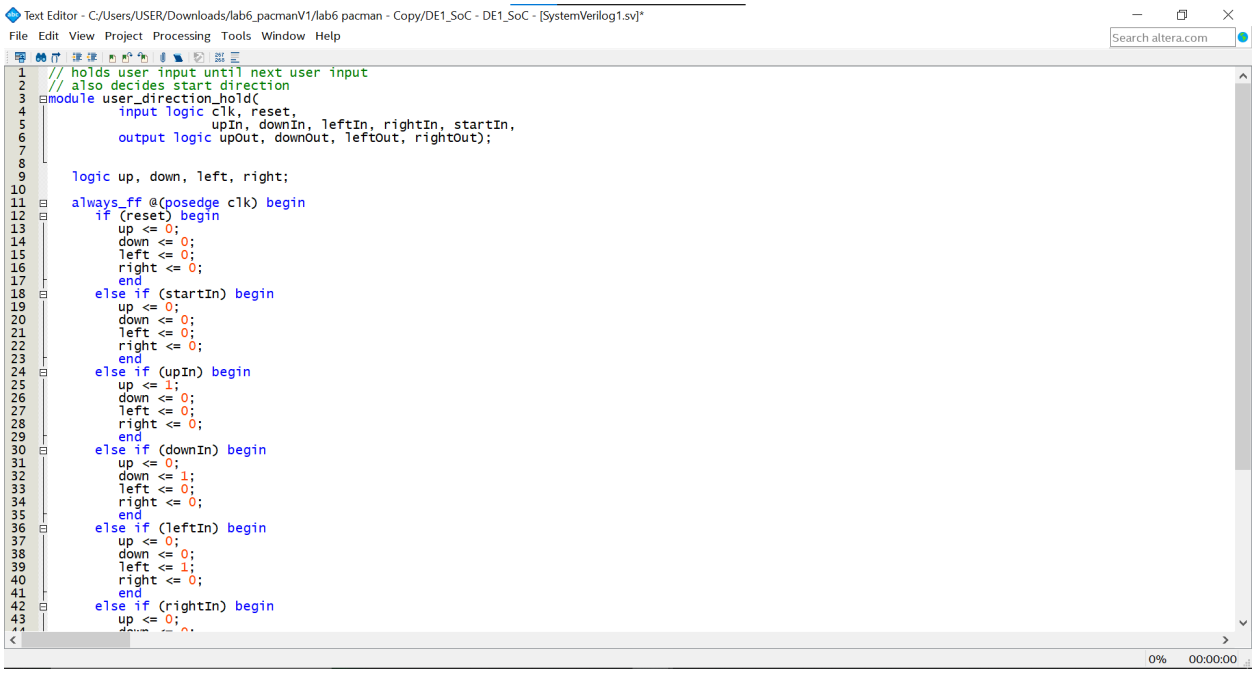


Figure 15. Systemverilog for user_direction_hold

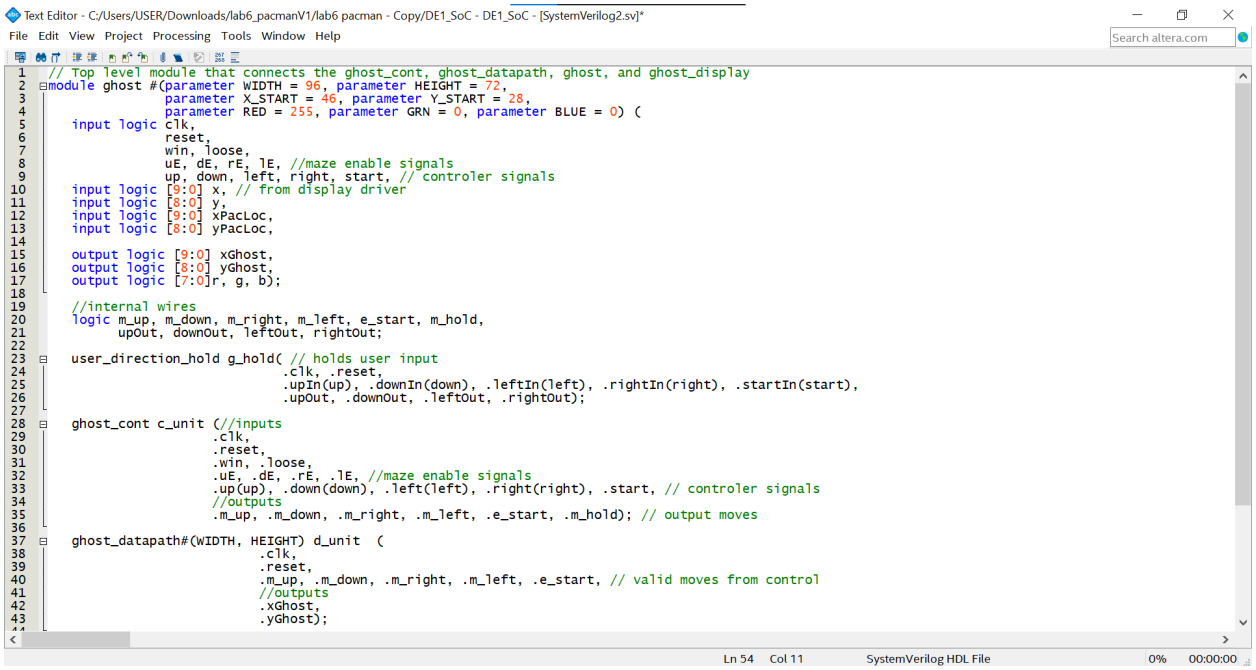


Figure 16. Systemverilog for ghost

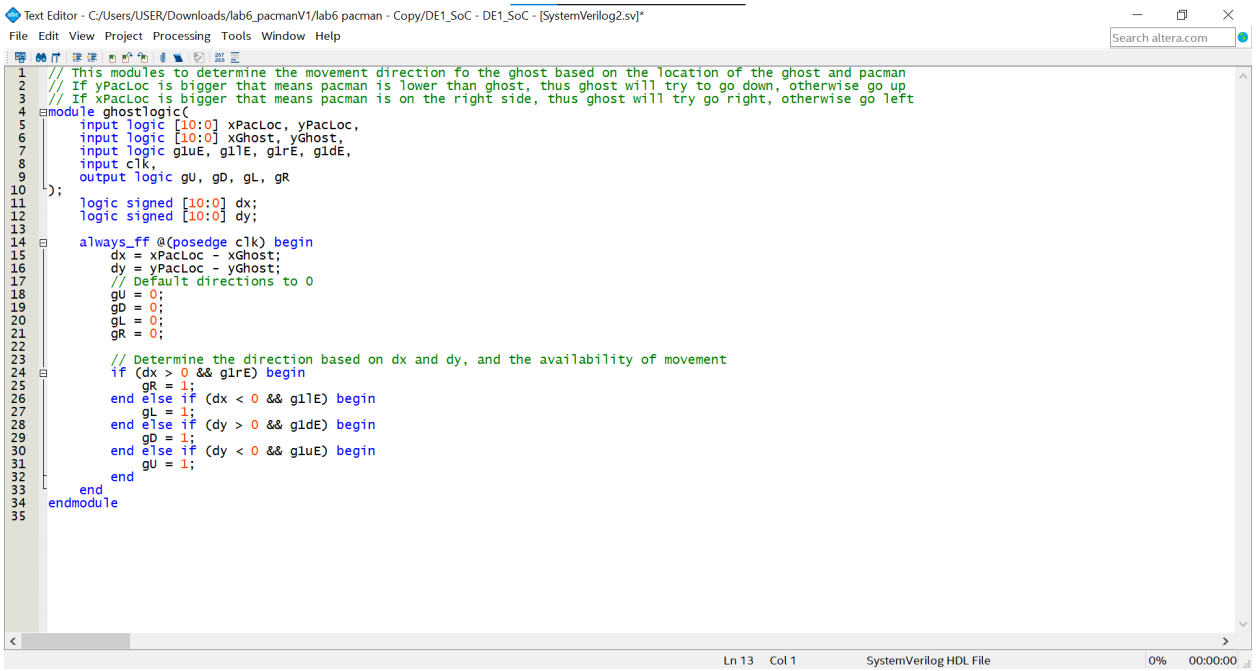


Figure 17. Systemverilog for ghostlogic

The implementation of the ghosts is similar to that of Pacman. However, instead of receiving direction signals from user input, the ghosts use a separate algorithm to determine their movement direction.

The algorithm determines the movement direction of the ghosts based on the relative positions of the ghost and Pacman. The logic is as follows:
Vertical Movement:

- If yPacLoc (Pacman's y-coordinate) is greater than the ghost's y-coordinate, it means Pacman is below the ghost. Therefore, the ghost will try to move down.
- If yPacLoc is less than or equal to the ghost's y-coordinate, Pacman is above the ghost. Thus, the ghost will try to move up.

Horizontal Movement:

- If xPacLoc (Pacman's x-coordinate) is greater than the ghost's x-coordinate, it means Pacman is to the right of the ghost. Therefore, the ghost will try to move right.
- If xPacLoc is less than or equal to the ghost's x-coordinate, Pacman is to the left of the ghost. Thus, the ghost will try to move left.

Thus, in this way, the ghost is always trying to chase the pacman until their locations are exactly the same.

Loser

After building the main components we are now ready to make a couple final touches to the game. We want to have significant things happen when the game is won or lost. For example, the whole display screen needs to change and display a message. We know if we have time, giving pacman extra lives would not be hard to implement given that he has his own reset button. Changing the whole display to say something like game over seemed like a necessary goal for a basic implementation. We knew that utilizing a ROM filled with the data would be one of the best ways to make this happen. See figure 18 below for details.

loser.sv*

```
1  // Displays GAME OVER based on Pacman and Ghost location
2  module loser#(parameter WIDTH = 96, parameter HEIGHT = 72)(
3
4      input logic clk, reset,
5      input logic [9:0] x, xPacLoc, xGhost,
6      input logic [8:0] y, yPacLoc, yGhost,
7
8      output logic gamelose,
9      output logic [7:0] r, g, b);
10
11
12     logic contact;
13     logic [12:0] displayAddress;
14     logic [7:0]  rOut, gOut, bOut;
15
16
17     // Remebers if pac & ghost meet
18     always_ff @(posedge clk)
19        if(reset)
20           contact = 0;
21        else if (xPacLoc == xGhost && yPacLoc == yGhost)
22           contact = 1;
23
24     // Converts x y location to address location
25     always_ff @(posedge clk)begin
26        if(reset)
27           displayAddress = 0;
28        else if (contact)begin
29           displayAddress <= (y * WIDTH + x) + 1;
30           end
31        end
32
33     ROM_gameOver_R
34        romR(
35              .address(displayAddress),
36              .clock(clk),
37              .q(rOut));
38
39
40     ROM_gameOver_G
41        romG(
42              .address(displayAddress),
43              .clock(clk),
44              .q(gOut));
45
46
47     ROM_gameOver_B
48        romB(
49              .address(displayAddress),
50              .clock(clk),
51              .q(bOut));
```

```
51                .q(bout));
52
53
54      // output logic
55      assign gamelose = contact;
56      assign r = rOut * 250;
57      assign g = gOut * 250;
58      assign b = bOut;
59
60
61  endmodule // loser
```

Figure 18. Systemverilog for loser

As we can see there are only a couple key points we needed to figure out.

- Trigger loose condition by contact of the ghost and pacman's location
- Convert the x and y coordinates to a ROM address

One lesson learned is that for our implementation we only needed one ROM due to the type of data available to generate for the MIF file. This is why the output values are scaled so great. More discussion on this in the results section.

Winner

The winner module and situation will use the framework of the loose situation but will be slimmed down due to not needing as many input signals. The win situation only is active when all the food is eaten so this single logic signal will come from the food module.

winner.sv* DE1_SoC.sv

```
1  // Winning display
2  // Starts reading ROM after winning condition met
3  module winner#(parameter WIDTH = 96, parameter HEIGHT = 72)(
4
5     input logic clk, reset, gamewin,
6     input logic [9:0] x,
7     input logic [8:0] y,
8
9     output logic [7:0] r, g, b);
10
11    logic contact;
12    logic [12:0] displayAddress;
13    logic [7:0]  qOut;
14
15    // Converts x y location to address location
16    always_ff @(posedge clk)begin
17       if(reset)
18          displayAddress = 0;
19       else if (gamewin)begin
20          displayAddress <= (y * WIDTH + x) + 1;
21
22          end
23       end
24
25    ROM_win
26       romW(
27             .address(displayAddress),
28             .clock(clk),
29             .q(qOut));
30
31    // output logic
32    assign gamelose = contact;
33    assign r = qOut * 250;
34    assign g = qOut * 250;
35    assign b = qOut;
36
37
38 endmodule // winner
39
```

Figure 19. Systemverilog for winner

As seen in figure 19 the winner module is just a slimmed down version of the loser module in figure 18. The ROM is at the heart of both the modules. The winner is looking for one signal to take over and celebrate the win.

Top Level module (DE1 - SoC)

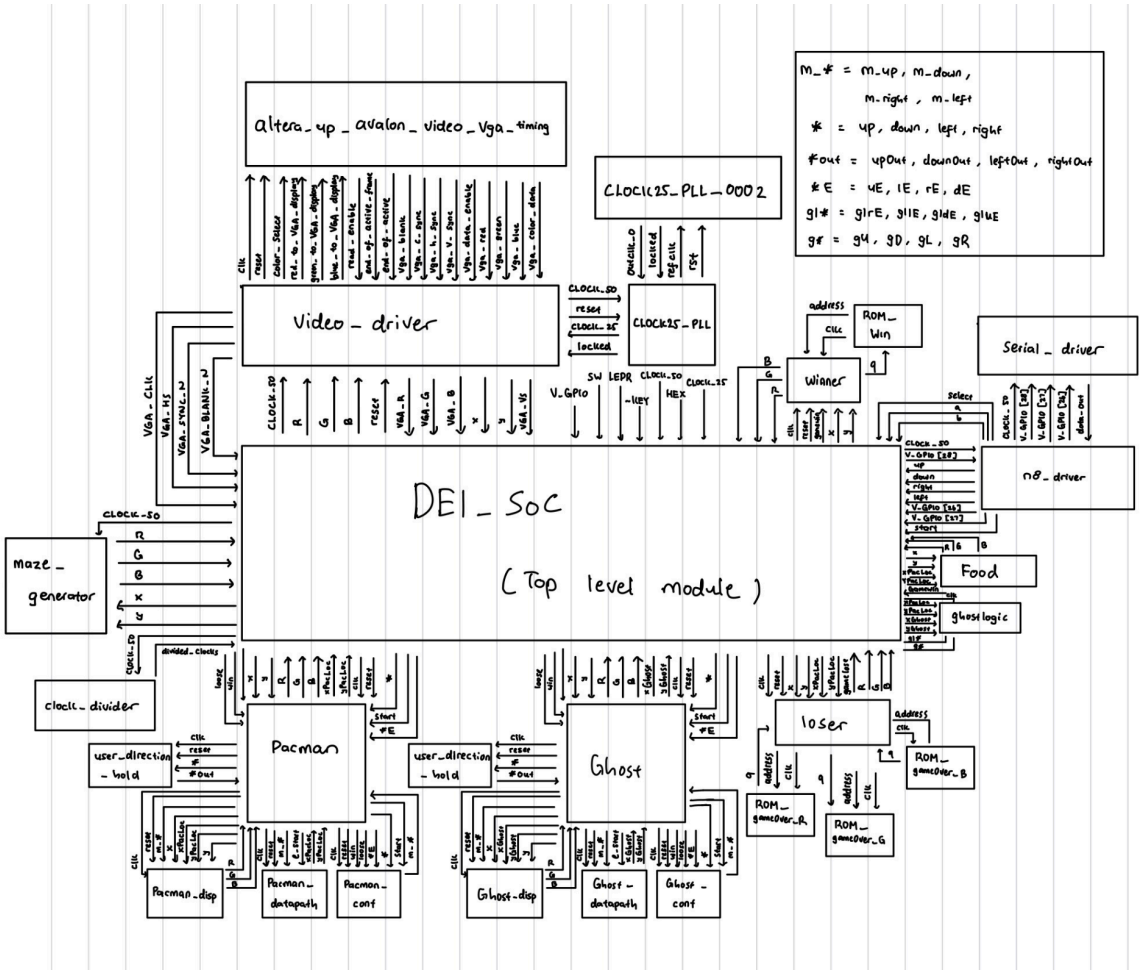


Figure 20. Block diagram of the top level

Once all the necessary modules were created, the next step was to integrate them into a top-level module, which connects the board and all the individual modules to enable the game to be played. Here's a more detailed breakdown of the integration process:

- Connecting the N8 Controller:
    - The N8 controller serves as the primary movement control signal for Pacman.
    - The up signal from the N8 controller makes Pacman move up.
    - The left signal makes Pacman move left.
    - The right signal makes Pacman move right.
    - The down signal makes Pacman move down.
    - Additionally, the start signal, also triggered from the N8 controller, initiates the game.
- Reset Signal:

    - The reset signal is controlled by ~KEY0. Pressing ~KEY0 will reset the game, returning Pacman and all game elements to their initial states.
- Integration of Modules: The top-level module integrates the maze, Pacman, ghosts, food, and VGA display modules.
- Maze Module: Defines the layout of the game area, including walls and pathways.
- Pacman Module: Controls Pacman's position, movement, and interactions with other game elements.
- Ghost Modules: Control the movement and behavior of ghosts, including their chasing logic and interaction with Pacman.
- Food Module: Manages the placement and consumption of food items by Pacman.
- VGA Display Module: Handles the graphical output to the display, rendering Pacman, ghosts, food, and the maze.
- Collision Detection and Game Logic:
    - The top-level module coordinates collision detection, ensuring Pacman interacts correctly with walls, food, and ghosts.
    - When Pacman collides with food, the food is consumed, and the score is updated.
    - If Pacman collides with a ghost, the game logic determines whether Pacman loses a life or if special conditions (e.g., power-ups) affect the interaction.
- Gameplay Flow:
    - At the start of the game, the start signal initializes all game elements.
    - Players use the N8 controller to navigate Pacman through the maze, avoiding ghosts and collecting food.
    - The reset signal can be used to restart the game at any time.

By integrating these components into a cohesive top-level module, we create a fully functional Pacman game that can be played on an FPGA with VGA output and N8 controller input. The top-level module ensures seamless interaction between all parts of the game, providing an engaging and responsive gaming experience.

Results

Our Pacman game turned out well with all the main features covered. The movement of Pacman simulated the original game play. This allows the user to push the direction once anytime before the turn and Pacman will make that turn. This all happened correctly within the maze boundaries.

The map is not identical to the original game but still has a similar feel. The food is another key part of the game that we were happy to get working properly. This took more than one try and ended up being the most cleverly written part of the code. Last but not least we also implemented the ghost. The logic for the ghost is a touch different then the original game. Basically our ghost chases Pacman much more from start to finish and never lets up. The framework of the code allows for multiple ghosts and multiple types of ghost logic. This game took lots of testing mostly through labsland but some through the testbench. In this section we will take a look at the game development and testing in the same order as it came together.

Maze

The maze was the beginning of using the new display driver. This set the tone for the rest of the game. One issue that we ran into is scaling. The maze code needed to be rewritten after we realized that life will be easier later on if we had a known set of pixels. We decided on a 96x72 frame to allow for our target size of 3x3 for Pacman. The maze turned out well and started our display logic. See figure 21 below for details

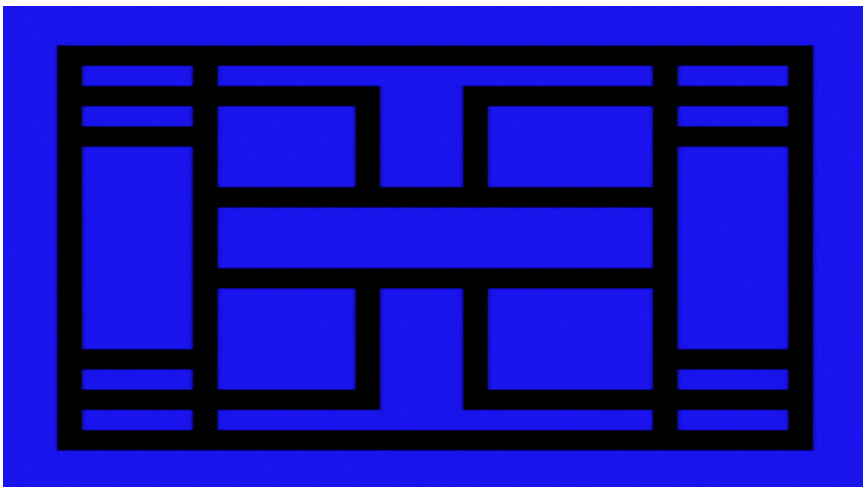


Figure 21. maze

Food

The food is a key part of the game and after understanding the display driver the food was implemented. One problem came about though because the food is not static like the maze it needs to change as Pacman eats it. This meant that the food code also needed to be rewritten. At

this time the focus shifted to getting Pacman moving to see if there are any other issues that we need to keep on radar for combining the modules. Below is the static food implementation that ultimately gets changed but keeps the same look.

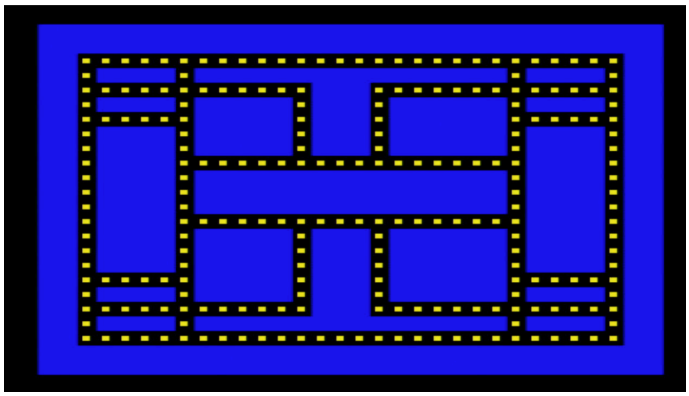


Figure 22. maze and food

Pacman and maze logic

Pacman is obviously the heart of the game so we wanted lots of control and interaction capabilities. This resulted in an AMSD design with lots of signals and outputs. To begin with we made it so that Pacman moves by pushing a button on the directional pad on the N8. This ended up working after a fair amount of struggle. Pacman was able to move anywhere on the screen, including through to the other side. We knew that in the game play of Pacman he needed to move on his own and just turn but we also needed to make a map logic that signaled to Pacman that he can not go through walls. So the map logic was made based on the map. With hindsight we wonder if there was a way to use the map as the map logic. It seems possible but tricky because the map is 3 pixels wide. The maze logic made it easy for Pacman to stay on track and know when there was a corner or turn. The maze logic worked well with the manual diving of Pacman. So then we moved to making Pacman move smoothly through the maze. This proved to be more difficult because of checking the next state. When Pacman was moved manually there was a hold state after each move, this allowed for enough time for the enable signals from the map logic to stop the movement. So when implementing the constant movement Pacman would step one pixel off and then be stuck because he was out of the maze logic. We ended up adding four more states to fix this timing issue. Each direction basically needed a check state which would pass through the movement if available. We needed to keep the hold state as well for when Pacman comes to a

stop after hitting a wall. The figure below shows a transition that is working properly but does not work for hitting walls.

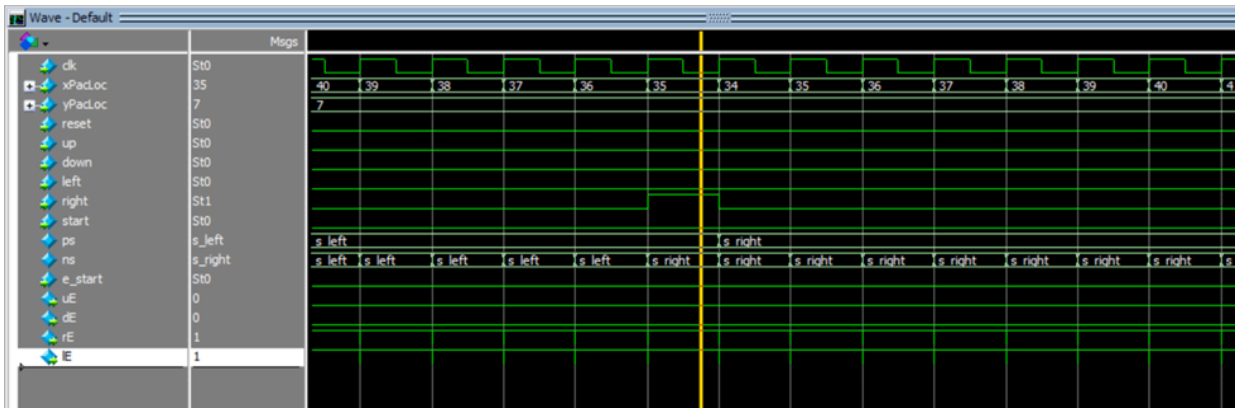


Figure 23. Pacman debugging waveform

We can see the x location change directions after the right signal. This works unless he hits a wall. Figure below is the corrected version. Here we can see the check states and the hold state when Pacman hits the wall.

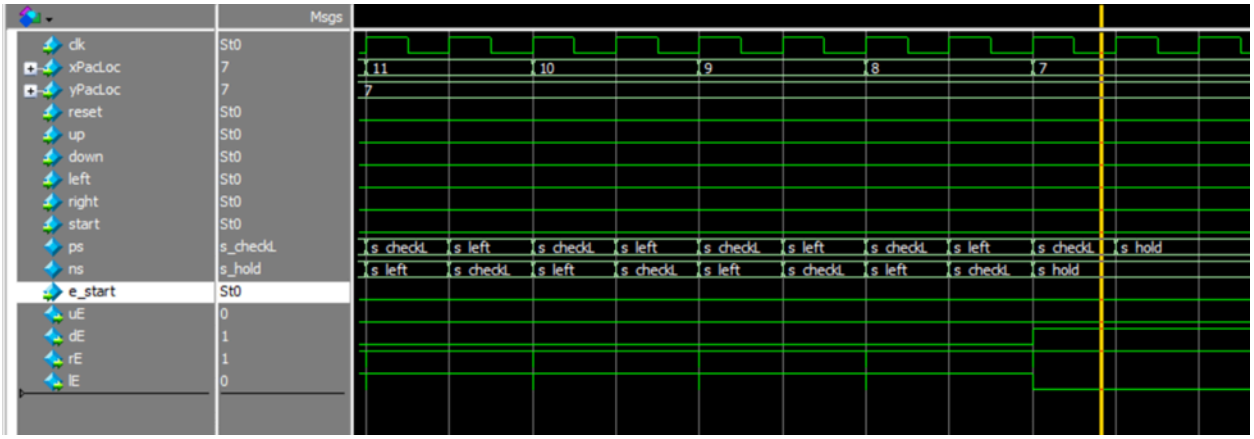


Figure 24. Pacman hitting wall waveform

Next issue for Pacman is remembering the last input signal. This was a relatively easy problem to solve with an additional module that connected to the inputs. It would only allow one signal to be transmitted at a time and then the signal would stay on until the other took over. See figure below for details. The waveform is not the best example but it does show that it is working. Basically the left signal is on and stays on. LeftIn is now the input and we can see it is only on

for a single clock where left is continually on. The reason this is not the best waveform is because the left signal is initially triggered at the start of the game. That's why it's on before leftIn is triggered.

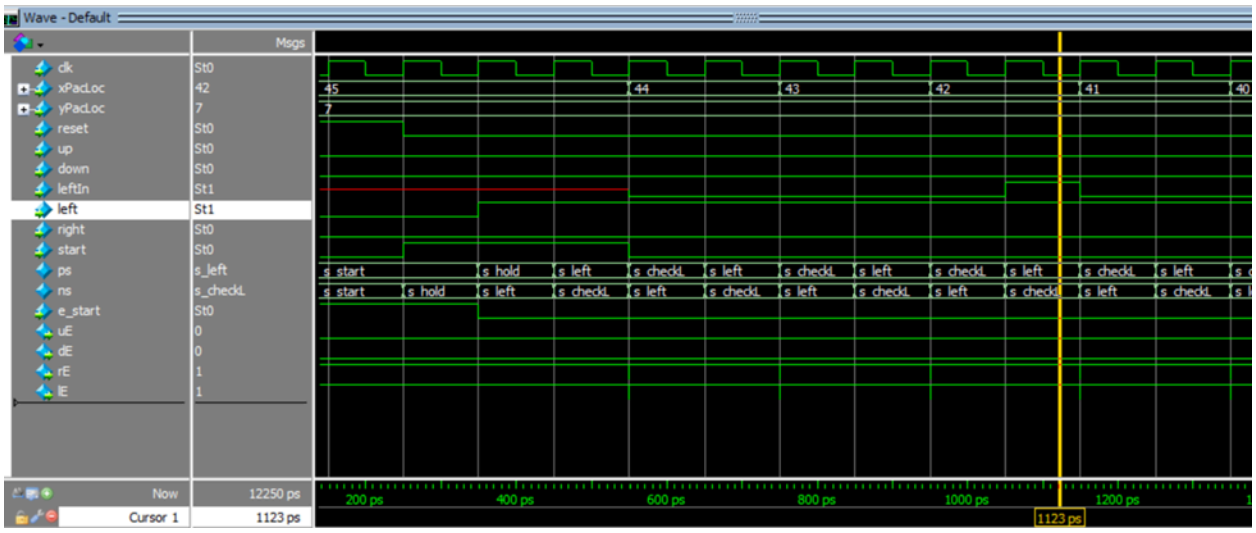


Figure 25. Pacman remember signals waveform

Now that Pacman can move and stop properly, final testing of his movement is done on the VGA display to ensure that nothing was missed. This is where we figured out a proper clock divider speed to be implemented. Pacman is the white cube in test mode. See figure below for details.

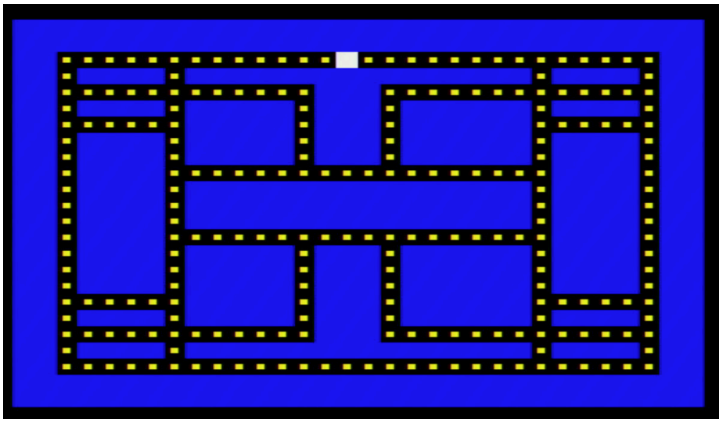


Figure 26. Pacman with movement in maze

Food

The new food module allowed Pacman to finally eat. Each piece of food now had a location in an array memory that would be updated when Pacmans location equaled the food location. This is a lot of locations so we implemented multiple loops to populate the different line locations.

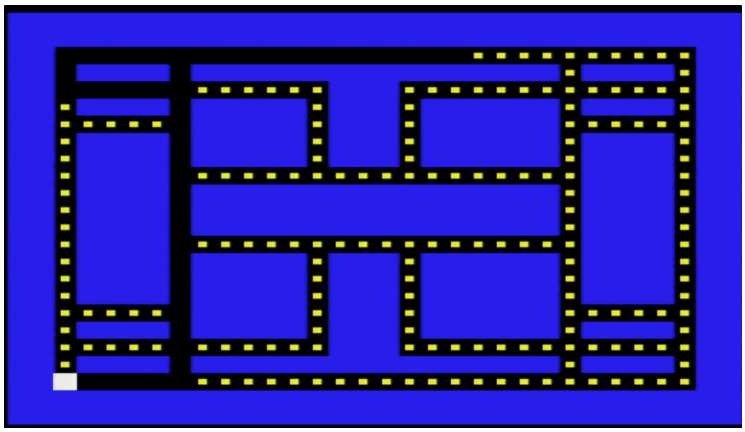


Figure 27. Game play with Pacman eating

Ghost

Pacman would not be fun without the ghosts. For our game we knew if we had Pacman working well that the ghost would be able to use most of the same logic and display modules. This turned out to be true. Only minor name changes were needed. The biggest issue was figuring out what clock should run which component. The one that caused the most trouble was the N8 controller. Even though the ghost does not use the controller, Pacman does and it needs to be at the same clock speed as he is running. We started the ghost the same way where we got it working with our inputs and then built the ghost logic to control after that.

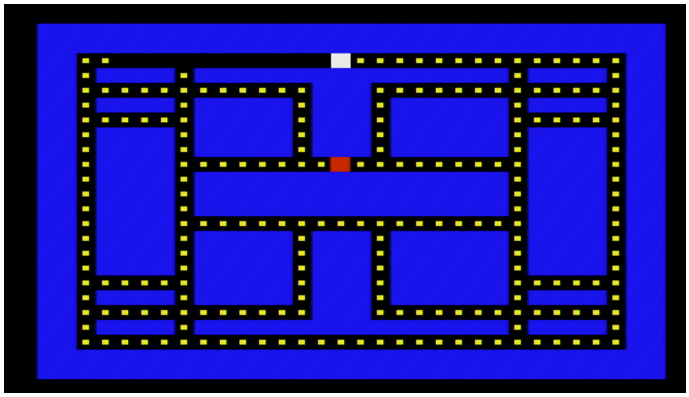


Figure 28. Pacman and ghost

Ghost Logic

The ghost logic determines the feel and stress of the game. Our ghost is able to see Pacmans location at all times. Then based on that location compared to the ghosts location the ghost only has one goal. Which is to be where Pacman is. This makes the game challenging even with only one ghost. If we had more time this is where we can imagine many kinds of ghost logic that could give the game different strategies and such. See figure 29 below for details.

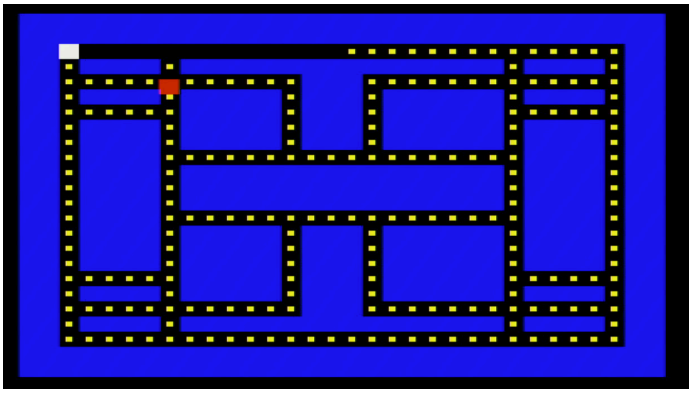


Figure 29. Ghost chasing Pacman

Loose

What happens when the Ghost catches Pacman? See figure 30.

Figure 30. Game Over ROM display

Once the Ghost catches Pacman the game is over and a ROM is then utilized to display this static image. The hardest part of using the ROM is just setting up a memory initialization file. This was possibly more difficult in our situation because of our unusual width and depth. We spent some time trying to figure out how to do an RGB image with multiple colors. This game over image is utilizing three ROMs but they all ended up having the same .mif file for simplicity.

Win

As Pacman was a learning experience for the Ghost the loose was for the win. The win logic comes from a slimmed down version of the loose module which then implements another ROM to display, see the image below in figure 31.

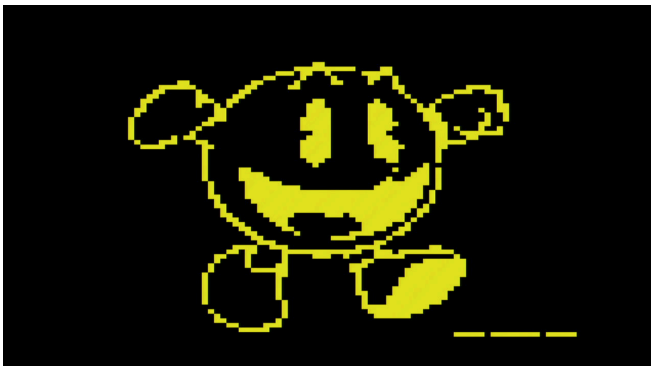


Figure 31. Winning ROM display