EE 354L Final Project Report
Spacewar!

Daniel Leng
Mark Camarena

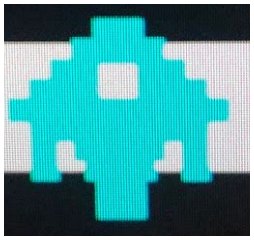
Description

Spacewar! is known for being one of the first computer games ever created in 1962. In homage to the original game, we decided to implement our own version of *Spacewar!* for the Nexys4 board. The goal of the game is to maneuver a spacecraft around falling asteroids and two flying saucers that actively target the player. The player must survive a total of ten rounds to win the game!

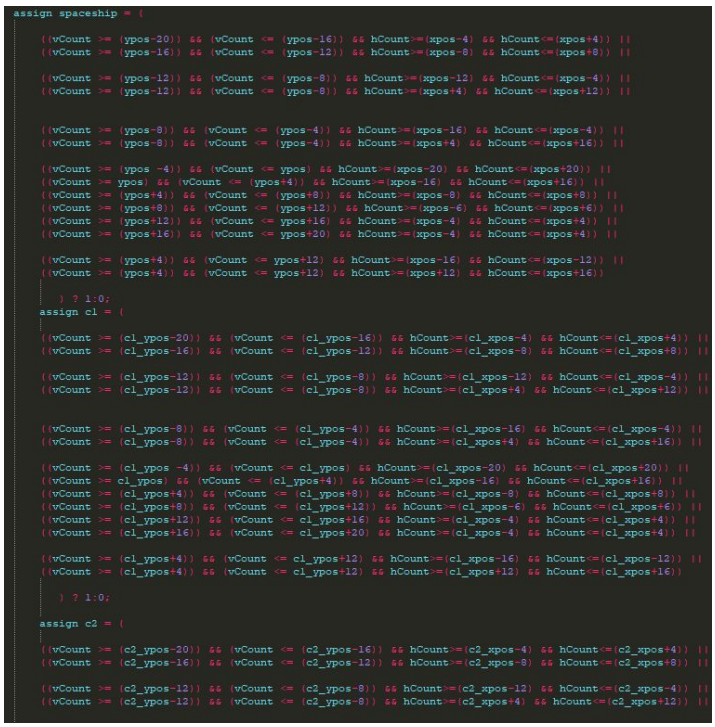
Design

Sprites

This is a game with spaceships and asteroids that are represented with small, simple sprites in the style of old Atari games. Our sprites are defined relative to one coordinate on the VGA display. The following is the sprite for our spaceship. The white bar behind the spaceship is not a part of the sprite.



This is a code snippet that defines the spaceship. It is quite long so here is an excerpt from the source code.



```
assign spaceship = (

    ((vCount >= (ypos-20)) && (vCount <= (ypos-16)) && hCount>=(xpos-4) && hCount<=(xpos+4)) ||
    ((vCount >= (ypos-16)) && (vCount <= (ypos-12)) && hCount>=(xpos-8) && hCount<=(xpos+8)) ||

    ((vCount >= (ypos-12)) && (vCount <= (ypos-8)) && hCount>=(xpos-12) && hCount<=(xpos-4)) ||
    ((vCount >= (ypos-12)) && (vCount <= (ypos-8)) && hCount>=(xpos+4) && hCount<=(xpos+12)) ||


    ((vCount >= (ypos-8)) && (vCount <= (ypos-4)) && hCount>=(xpos-16) && hCount<=(xpos-4)) ||
    ((vCount >= (ypos-8)) && (vCount <= (ypos-4)) && hCount>=(xpos+4) && hCount<=(xpos+16)) ||

    ((vCount >= (ypos -4)) && (vCount <= ypos) && hCount>=(xpos-20) && hCount<=(xpos+20)) ||
    ((vCount >= ypos) && (vCount <= (ypos+4)) && hCount>=(xpos-16) && hCount<=(xpos+16)) ||
    ((vCount >= (ypos+4)) && (vCount <= (ypos+8)) && hCount>=(xpos-8) && hCount<=(xpos+8)) ||
    ((vCount >= (ypos+8)) && (vCount <= (ypos+12)) && hCount>=(xpos-6) && hCount<=(xpos+6)) ||
    ((vCount >= (ypos+12)) && (vCount <= ypos+16) && hCount>=(xpos-4) && hCount<=(xpos+4)) ||
    ((vCount >= (ypos+16)) && (vCount <= ypos+20) && hCount>=(xpos-4) && hCount<=(xpos+4)) ||

    ((vCount >= (ypos+4)) && (vCount <= ypos+12) && hCount>=(xpos-16) && hCount<=(xpos-12)) ||
    ((vCount >= (ypos+4)) && (vCount <= ypos+12) && hCount>=(xpos+12) && hCount<=(xpos+16))

      ) ? 1:0;
  assign c1 = (

  ((vCount >= (c1_ypos-20)) && (vCount <= (c1_ypos-16)) && hCount>=(c1_xpos-4) && hCount<=(c1_xpos+4)) ||
  ((vCount >= (c1_ypos-16)) && (vCount <= (c1_ypos-12)) && hCount>=(c1_xpos-8) && hCount<=(c1_xpos+8)) ||

  ((vCount >= (c1_ypos-12)) && (vCount <= (c1_ypos-8)) && hCount>=(c1_xpos-12) && hCount<=(c1_xpos-4)) ||
  ((vCount >= (c1_ypos-12)) && (vCount <= (c1_ypos-8)) && hCount>=(c1_xpos+4) && hCount<=(c1_xpos+12)) ||


  ((vCount >= (c1_ypos-8)) && (vCount <= (c1_ypos-4)) && hCount>=(c1_xpos-16) && hCount<=(c1_xpos-4)) ||
  ((vCount >= (c1_ypos-8)) && (vCount <= (c1_ypos-4)) && hCount>=(c1_xpos+4) && hCount<=(c1_xpos+16)) ||

  ((vCount >= (c1_ypos -4)) && (vCount <= c1_ypos) && hCount>=(c1_xpos-20) && hCount<=(c1_xpos+20)) ||
  ((vCount >= c1_ypos) && (vCount <= (c1_ypos+4)) && hCount>=(c1_xpos-16) && hCount<=(c1_xpos+16)) ||
  ((vCount >= (c1_ypos+4)) && (vCount <= (c1_ypos+8)) && hCount>=(c1_xpos-8) && hCount<=(c1_xpos+8)) ||
  ((vCount >= (c1_ypos+8)) && (vCount <= (c1_ypos+12)) && hCount>=(c1_xpos-6) && hCount<=(c1_xpos+6)) ||
  ((vCount >= (c1_ypos+12)) && (vCount <= c1_ypos+16) && hCount>=(c1_xpos-4) && hCount<=(c1_xpos+4)) ||
  ((vCount >= (c1_ypos+16)) && (vCount <= c1_ypos+20) && hCount>=(c1_xpos-4) && hCount<=(c1_xpos+4)) ||

  ((vCount >= (c1_ypos+4)) && (vCount <= c1_ypos+12) && hCount>=(c1_xpos-16) && hCount<=(c1_xpos-12)) ||
  ((vCount >= (c1_ypos+4)) && (vCount <= c1_ypos+12) && hCount>=(c1_xpos+12) && hCount<=(c1_xpos+16))

    ) ? 1:0;

  assign c2 = (

  ((vCount >= (c2_ypos-20)) && (vCount <= (c2_ypos-16)) && hCount>=(c2_xpos-4) && hCount<=(c2_xpos+4)) ||
  ((vCount >= (c2_ypos-16)) && (vCount <= (c2_ypos-12)) && hCount>=(c2_xpos-8) && hCount<=(c2_xpos+8)) ||

  ((vCount >= (c2_ypos-12)) && (vCount <= (c2_ypos-8)) && hCount>=(c2_xpos-12) && hCount<=(c2_xpos-4)) ||
  ((vCount >= (c2_ypos-12)) && (vCount <= (c2_ypos-8)) && hCount>=(c2_xpos+4) && hCount<=(c2_xpos+12)) ||
```

The important takeaway from this is that `hCount` and vCount govern which pixels are colored in (not black) to display the sprite. If either of these variables reach parts of the display that fall within the range of `xpos` and `ypos` (xpos and ypos are the x and y coordinates of the spaceship used throughout our code).

Asteroids are defined as rectangles similarly to the spaceship. Had there not been any time constraints, we would have designed more sophisticated sprites for the asteroids.

State Machine

The game's logic and some of the visual effects used on the VGA display are governed by our state machine.

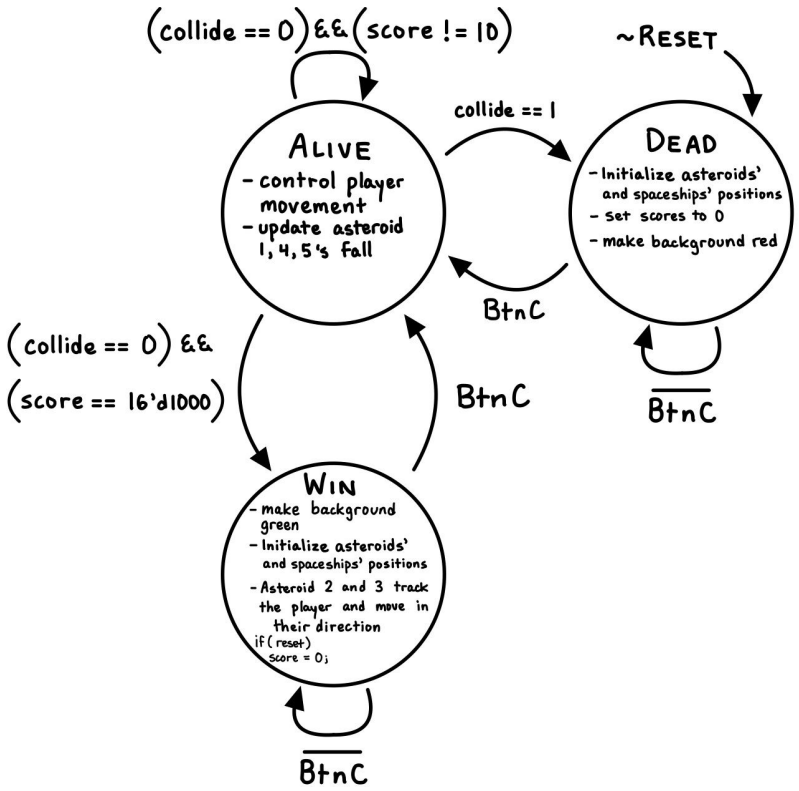


Figure 1. Spacewar! State Diagram

In our source code, our RTL implementations are split across multiple clock and reset-controlled `always@()` blocks for organization. The following is a higher-level explanation of each state's functions accompanied with snippets of our source code:

```
// State Machine and Next State Logic
always@(posedge clk, posedge rst)
    begin
        if(rst)
            state = ALIVE;
        else if(clk)
            begin
                case(state)
                    ALIVE:
                    begin
                        background <= BLACK;
                        if(collide)
                            state <= DEAD;
                        else if(score == 16'd1000)
                            state <= WIN;
                    end
                    WIN:
                    begin
                        background <= GREEN;
                    end
                    DEAD:
                    begin
                        background <= RED;
                    end
                    default: background <=BLACK;
                endcase
            end
    end
```

Figure 2. Verilog Code Implementing State Machine

ALIVE: This state handles most of our game's core functionality such as controlling the player sprite, updating the location of the falling and homing asteroids along their trajectories, and keeping track of their score. The score increments as time progresses and the player has not collided with anything. The score is outputted to four digits as a decimal value on the seven-segment display. This state will first set the background of the VGA display to black. It then proceeds to start the multiple rounds of the game. With each new round in the ALIVE state, we are checking if the player collides with any object (asteroid or spaceship). If there is a collision, the player is sent to the DEAD state. If the player successfully completes the rounds and achieves a score of 1000, the player is sent to the WIN state indicating they have won the game.

```
ALIVE:
begin
    background <= BLACK;
    if(collide)
        state <= DEAD;
    else if(score == 16'd1000)
        state <= WIN;
end
```

Figure 3. ALIVE state

Player Movement

As referenced in the state diagram, this state is also responsible for enabling control of the player as well as controlling all enemy objects. We have an `xpos` and `ypos` describing the player's coordinates on the VGA display at any given time.

The convention we used to define the two-dimensional coordinate axes for the VGA display is to set the origin of the plane in the upper left corner of the display where anything below the origin is defined as "positive" for `ypos` and anything to the right of the origin is "positive" for `xpos`.

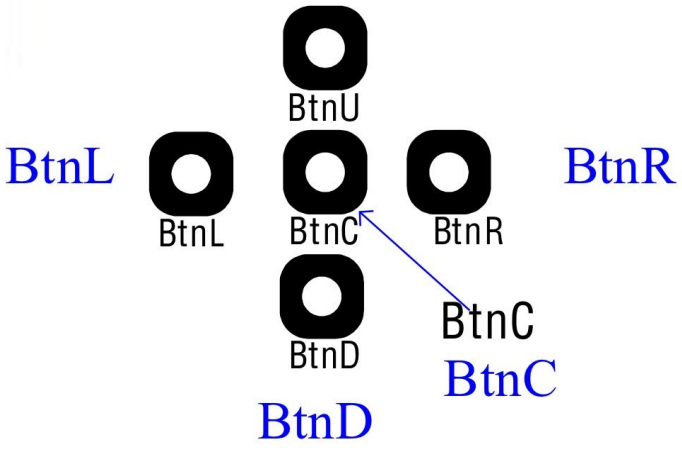


Figure 4. Button Layout on Nexys4

For our player, decrementing the value of `xpos` makes the player move towards the left of the screen. Incrementing `xpos` makes the player move to the right. In our source code, we are using **BtnL** and **BtnR** on the Nexys4 boards as inputs for manipulating the values of the `xpos` and

therefore, the player's position on the VGA display. Pressing BtnL moves the player to the left and BtnR moves to the right.

```
else if (state == ALIVE) begin

    if(right) begin
        xpos<=xpos+shipSpeed; //change the amount you increment to make the speed faster
        if(xpos==776) //these are rough values to attempt looping around,
    end
    else if(left) begin
        xpos<=xpos-shipSpeed;
        if(xpos==152)
            xpos<=152;
    end
end
end
```

Figure 5. Excerpt from "Pixel Position" always@() block

This code snippet is what actually defines the player ship's movements. The input signals `left` and `right` come from the top design file. BtnL and BtnR activate `left` and `right` respectively (see below for more information). The integer `shipSpeed` is set to 4 which is fast enough for the player to avoid enemies while still keeping the game slightly challenging. The nested if statements are there to check and see if the ship is at either edge of the VGA display. These specific values were used because it allows the player sprite to stay on screen at all times.
Asteroid Movement:

There are two types of asteroids in the game: falling asteroids and homing asteroids. Each round features three different falling asteroids and two homing asteroids. The falling asteroids (defined as Asteroids 1, 4 and 5 in the source code) just fall vertically in a straight line.

```
else if(state == ALIVE) begin
    if(!collide)begin
            asteroidY_1 <= asteroidY_1 + asteroid_1Speed;
        if(asteroidY_1 >= 10'd 779) begin
            asteroidY_1 <= 0;
        end
    end
end
```

Figure 6. Example of a Falling Asteroid's Movement

The code snippet above shows the behavior of asteroid 1. As long as no collision occurs, the asteroid will keep "falling" down towards the player. Falling is achieved by incrementing `asteroidY_1` (Asteroid 1's y-coordinate) by `asteroid_1Speed` which is an integer that controls the speed. Very much like player movement, the larger the "speed" value, the more distance the asteroid will cover in a single clock. This gives the sense that the asteroid is

“falling” at a quicker rate. Different falling asteroids use different speed values, as shown in the screenshot below.

```
wire asteroid_4;
reg[9:0] asteroidY_4;
reg[9:0] asteroidX_4;
integer asteroid_4Speed = 1;
integer a_4Size = 40;

wire asteroid_5;
reg[9:0] asteroidY_5;
reg[9:0] asteroidX_5;
integer asteroid_5Speed = 3;
integer a_5Size = 40;
```

The homing asteroids, asteroids 2 and 3, work a little differently. Let’s look at Asteroid 2 as an example. The exact same idea applies to the other homing asteroid, Asteroid 3.

```
//asteroid 2
always@(posedge clk)
    begin
        if (clk)begin
            if((state == DEAD) || (state == WIN))begin
                asteroidY_2 <= 10'd50;
                asteroidX_2 <= 10'd800;
            end
            else if(state == ALIVE) begin
                asteroidY_2 <= asteroidY_2 + asteroid_2Speed/10;
                asteroidX_2 <= asteroidX_2 - a_2xtraj/10;
                if(asteroidY_2 >= 10'd779) begin
                        asteroidY_2 <= 0;
                        asteroidX_2 <= 800;
                end
            end
        end
    end
```

Figure 7. Homing Asteroid 2’s Movement

The homing asteroid moves vertically downwards but we also use two tasks to calculate the trajectory of each homing asteroid. The homing asteroids start at opposite ends of the VGA display and adjust their trajectories based on the last known coordinates of the player ship. The following code demonstrates how we generate the asteroids’ trajectories.

```
task traj_gen_right;
    input[9:0] aX, aY, pX, pY;
    input integer speed;
    output integer value;
    integer ax,ay,px,py,xdiff,ydiff,x;
    begin
        ax = aX;
        ay = aY;
        px = pX;
        py = pY;
        xdiff = ax-px;
        ydiff = py-ay;
        x = speed*xdiff/ydiff;
        value = x;
    end
endtask
    task traj_gen_left;
    input[9:0] aX, aY, pX, pY;
    input integer speed;
    output integer value;
    integer ax,ay,px,py,xdiff,ydiff,x;
    begin
        ax = aX;
        ay = aY;
        px = pX;
        py = pY;
        xdiff = px-ax;
        ydiff = py-ay;
        x = speed*xdiff/ydiff;
        value = x;
    end
endtask
```

Figure 8. Trajectory Calculations

In Figure 8, the tasks `traj_gen_right` and `traj_gen_left` take in `aX`, `aY`, `pX` and `pY`. Tasks are Verilog's version of functions. The task is passed in `aX` and `aY` which represent an asteroid's x-coordinate and y-coordinate respectively. `pX` and `pY` represent the player ship's x and y coordinates respectively. Task `traj_gen_right` will then find the difference of `aX` - `pX` and task `traj_gen_left` will find the difference of pX - aX. They then each divide either `aX` - `pX` or `pX` - `aX` by `pY` - `aY` depending on the task. The quotient is then multiplied by `asteroidSpeed` which was passed in (it should not change in the rest of the code).

For Asteroids moving to the *left* approaching from the *right*, we use `trac_gen_right` where the trajectory is defined as:

$$\text{Trajectory}_{\text{right}} = \left(\frac{aX - pX}{pY - aY}\right) * (\text{speed})$$

For Asteroids moving to the *right* approaching from the *left*, we use `trac_gen_left` where the trajectory is defined as:

$$\text{Trajectory}_{\text{right}} = \left(\frac{pX - aX}{pY - aY}\right) * (\text{speed})$$

This is really just a simple "rise over run" slope calculation. It is visualized below with this annotated screenshot from the .bit file! "New Speed" refers to $\text{Trajectory}_{\text{speed}}$ calculated by `trac_gen_right.`

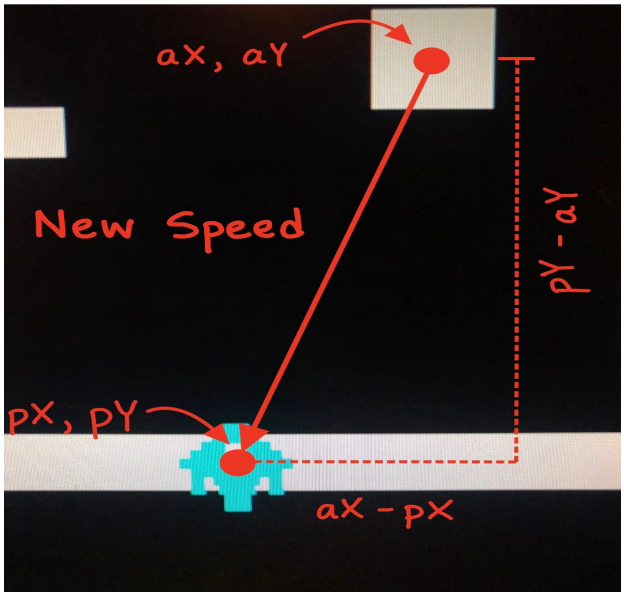


Figure 9. Visual Demonstration of Calculating Trajectory, Screenshot from .bit file

Collision Detection:

In this ALIVE state, we are also checking for any collisions between the player ship and any one of the five incoming asteroids. In this implementation, we are using the same principle to detect each collision and the same process is repeated for each asteroid.

Because the asteroids are coming at the spaceship from the top of the screen, we prioritized implementing collisions between the spaceship and the bottom of all of the asteroids. The diagram helps illustrate some of the comparisons made for collision detection. Because we can only work with the central coordinates of the spaceship and asteroids, we have to create a "hitbox" around (pX, pY) and (aX, aY) which is a region of coordinates surrounding the central coordinates. If the hitboxes of an asteroid and the spaceship intersect or overlap then a collision must have occurred.

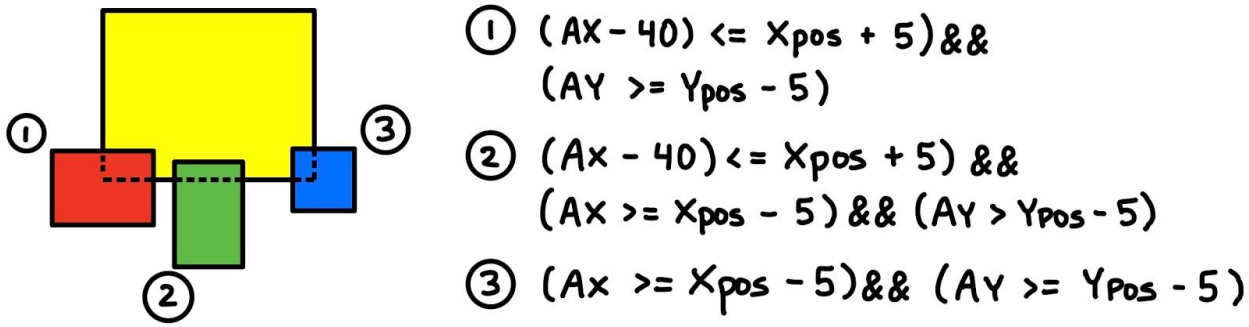


Figure 10. Visualization of Collision Detection with a 40-pixel asteroid

```
//collision detection
always@(posedge clk)
  begin
    if (clk) begin
      if ((state == DEAD) || (state == WIN))
        collide <= 1'b0;
      else if (state == ALIVE) begin
        if(
        (((asteroidX_1 - a_1Size*2) <= (xpos+shipSize)) && ((asteroidX_1 - a_1Size*2) >= (xpos-shipSize)) && (asteroidY_1 >= (ypos-shipSize)) && (asteroidY_1-a_1Size/2 <= (ypos+shipSize))) ||
        (((asteroidX_1 - a_1Size*2) <= (xpos+shipSize)) && (asteroidX_1 >= (xpos -shipSize)) && (asteroidY_1 >= (ypos-shipSize)) && (asteroidY_1-a_1Size/2 <= (ypos+shipSize))) ||
        ((asteroidX_1 >= (xpos-shipSize))&& ((asteroidX_1 - a_1Size*2) <= (xpos+shipSize)) && (asteroidY_1 >= (ypos-shipSize)) && (asteroidY_1-a_1Size/2 <= (ypos+shipSize))) ||

        (((asteroidX_2 - a_2Size) <= (xpos+shipSize)) && ((asteroidX_2 - a_2Size) >= (xpos-shipSize)) && (asteroidY_2 >= (ypos-shipSize)) && (asteroidY_2-a_2Size <= (ypos+shipSize))) ||
        (((asteroidX_2 - a_2Size) <= (xpos+shipSize)) && (asteroidX_2 >= (xpos -shipSize)) && (asteroidY_2 >= (ypos-shipSize)) && (asteroidY_2-a_2Size <= (ypos+shipSize))) ||
        ((asteroidX_2 >= (xpos-shipSize))&& ((asteroidX_2 - a_2Size) <= (xpos+shipSize)) && (asteroidY_2 >= (ypos-shipSize)) && (asteroidY_2-a_2Size <= (ypos+shipSize))) ||

        (((asteroidX_3 - a_3Size) <= (xpos+shipSize)) && ((asteroidX_3 - a_3Size) >= (xpos-shipSize)) && (asteroidY_3 >= (ypos-shipSize)) && (asteroidY_3-a_3Size <= (ypos+shipSize))) ||
        (((asteroidX_3 - a_3Size) <= (xpos+shipSize)) && (asteroidX_3 >= (xpos -shipSize)) && (asteroidY_3 >= (ypos-shipSize)) && (asteroidY_3-a_3Size <= (ypos+shipSize))) ||
        ((asteroidX_3 >= (xpos-shipSize))&& ((asteroidX_3 - a_3Size) <= (xpos+shipSize)) && (asteroidY_3 >= (ypos-shipSize)) && (asteroidY_3-a_3Size <= (ypos+shipSize))) ||

        (((asteroidX_4 - a_4Size*4) <= (xpos+shipSize)) && ((asteroidX_4 - a_4Size*4) >= (xpos-shipSize)) && (asteroidY_4 >= (ypos-shipSize)) && (asteroidY_4-a_4Size/2 <= (ypos+shipSize))) ||
        (((asteroidX_4 - a_4Size*4) <= (xpos+shipSize)) && (asteroidX_4 >= (xpos -shipSize)) && (asteroidY_4 >= (ypos-shipSize)) && (asteroidY_4-a_4Size/2 <= (ypos+shipSize))) ||
        ((asteroidX_4 >= (xpos-shipSize))&& ((asteroidX_4 - a_4Size*4) <= (xpos+shipSize)) && (asteroidY_4 >= (ypos-shipSize)) && (asteroidY_4-a_4Size/2 <= (ypos+shipSize)))||

        (((asteroidX_5 - a_5Size*2) <= (xpos+shipSize)) && ((asteroidX_5 - a_5Size*2) >= (xpos-shipSize)) && (asteroidY_1 >= (ypos-shipSize)) && (asteroidY_5-a_5Size/2 <= (ypos+shipSize))) ||
        (((asteroidX_5 - a_5Size*2) <= (xpos+shipSize)) && (asteroidX_5 >= (xpos -shipSize)) && (asteroidY_5 >= (ypos-shipSize)) && (asteroidY_5-a_5Size/2 <= (ypos+shipSize))) ||
        ((asteroidX_5 >= (xpos-shipSize))&& ((asteroidX_5 - a_5Size*2) <= (xpos+shipSize)) && (asteroidY_5 >= (ypos-shipSize)) && (asteroidY_5-a_5Size/2 <= (ypos+shipSize)))
        ) begin
          collide <= 1'b1;
        end
        end
      end
  end
```

Figure 11. Collision Detection Implemented

In this `always@()` block used for collision detection, we simultaneously check if the spaceship has collided with any one of the five asteroids.

DEAD:

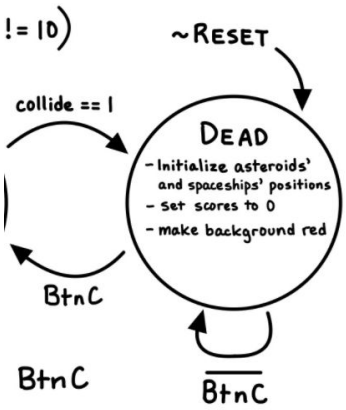


Figure 12. The Dead State from the State Machine

In the dead state, the state machine will set the background color to red, reset the positions of asteroids and the spaceship, and reset the score to 0.

```
if(clk)begin
    if((state == DEAD) || (state == WIN))begin
        traj_gen_right(asteroidX_2,asteroidY_2,xpos,ypos,asteroid_2Speed,a_2xtraj);
        traj_gen_left(asteroidX_3,asteroidY_3,xpos,ypos,asteroid_3Speed,a_3xtraj);
```

Figure 13. Calculating Trajectories within the Dead and Win States

In the dead state, we also calculate the trajectories of both homing asteroids so that they are ready for the next game!

When collisions are detected, the game uses a single-bit collision flag, `collide`, to signal to the state machine to transition states and govern the behavior of asteroids. Once in the dead state or win state, the collision flag is then set to 0.

```
//collision detection
always@(posedge clk)
    begin
        if (clk) begin
            if ((state == DEAD) || (state == WIN))
                collide <= 1'b0;
```

Figure 14. Turning off the Collision flag

```
    DEAD:
    begin
        background <= RED;
    end
    default: background <=BLACK;
```

Figure 15. Setting the background Red

The dead state also takes care of setting the screen to red and setting the score to 0 to indicate to the player that they are dead.

WIN:

The win state is greatly similar to the Dead State. There are many instances where certain code intended for the Dead state also runs when the state machine is in the Win State. This was all by design. An example of this can be seen in Figure 13 above.

The key differences between the win state and the dead state are:

- The win state uses a green background and shows three spaceship sprites. If the player sees the green background, they have won!
- The score does not reset when entering the win state. This was originally a bug, but we decided that keeping the score actually makes players look more impressive :)

I/O

Inputs

Name	Description
Clk	The clock signal that drives the system and the frequency at which objects display on the screen
bright	This signal indicates whether a given pixel is “bright” i.e. the monitor is outputting actual data or not. Used so that the monitor always outputs something to every pixel.
up	Input signal being passed in from the top design. This signal goes high upon BtnU
down	Input signal being passed in from the top design. This signal goes high upon BtnD
left	Input signal being passed in from the top design. This signal goes high upon BtnL
right	Input signal being passed in from the top design. This signal goes high upon BtnR
hCount	Keeps track of the horizontal coordinate of the pixel being written onto the VGA display
vCount	Keeps track of the vertical coordinate of the pixel being written onto the VGA display

NOTE: In our final game, we did not actually use the “up” and “down” signals. They are there because we were planning on adding different features.

Outputs

Name	Description
score	Score value is outputted from the core file and sent to the SSD
rgb	This outputs the sprites and horizontal vertical line to VGA.
background	This is what allows us to change colors of the background by adjusting the 12-bit RGB value.

Challenges

Tuning "Rate" of Asteroid Homing

One of the greater challenges was developing the honing mechanism for the asteroids. We needed to find a systematic way for asteroids to efficiently and accurately create a path to hit the spaceship wherever the spaceship may be on the screen. We did not want to use too many resources nor spend too much time computing stuff in a clock. We found the rise over run to work really well but also very easy to explain and understand. By using speed and trajectory variables, we were able to move objects diagonally.

Tasks

In order to implement the trajectory calculations, we had to use tasks. Starting this project, we lacked experience in designing Verilog tasks. Understanding the quirks of Verilog functions and its syntax was a challenge. We had to learn how to pass in different variables/registers and how to use the values that these tasks produced.

Improvements

Due to time constraints, we were unable to develop all the features we planned to implement. For example, the classic Spacewar contains 10 levels of increasing difficulty. Initially we planned to design our one-hot state machine with increasing levels of difficulty. Some other features were as follows:

- Graphics & Animation
    - Add color blocks to pixel art
    - Animate sprites
    - Splash screen to introduce the player to the game
- Boss Battle
    - With "homing missiles" and other "power ups," it would've been fun to design a boss battle type stage.

Our core design module could be modularized for improved readability and usability. The core design consisted of the state machine, next state logic, initialization of sprites, trajectory generation, obstacle detection, and other elements–most of which could be abstracted and organized into separate files.