COMBINATIONAL MAGIC PRESENTS

DRAMATIC MUSIC!!!!

Tilt

By Aymeric Blaizot, Tom Barrette, Thomas Young, Connor Casey

General Overview

Tilt is a maze solving game using motion controls.

Core Parts:

VGA Display

Accelerometer

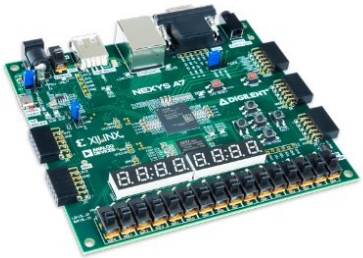
Collision Checking

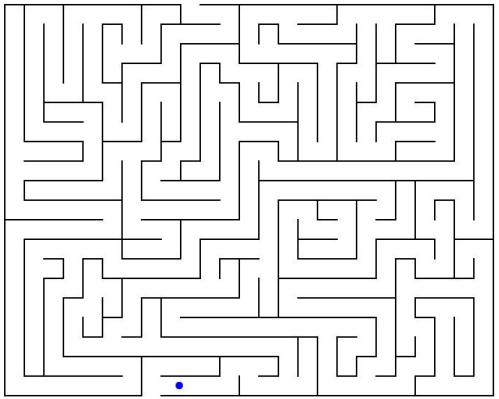
Seven Segment Display

Maze Generation

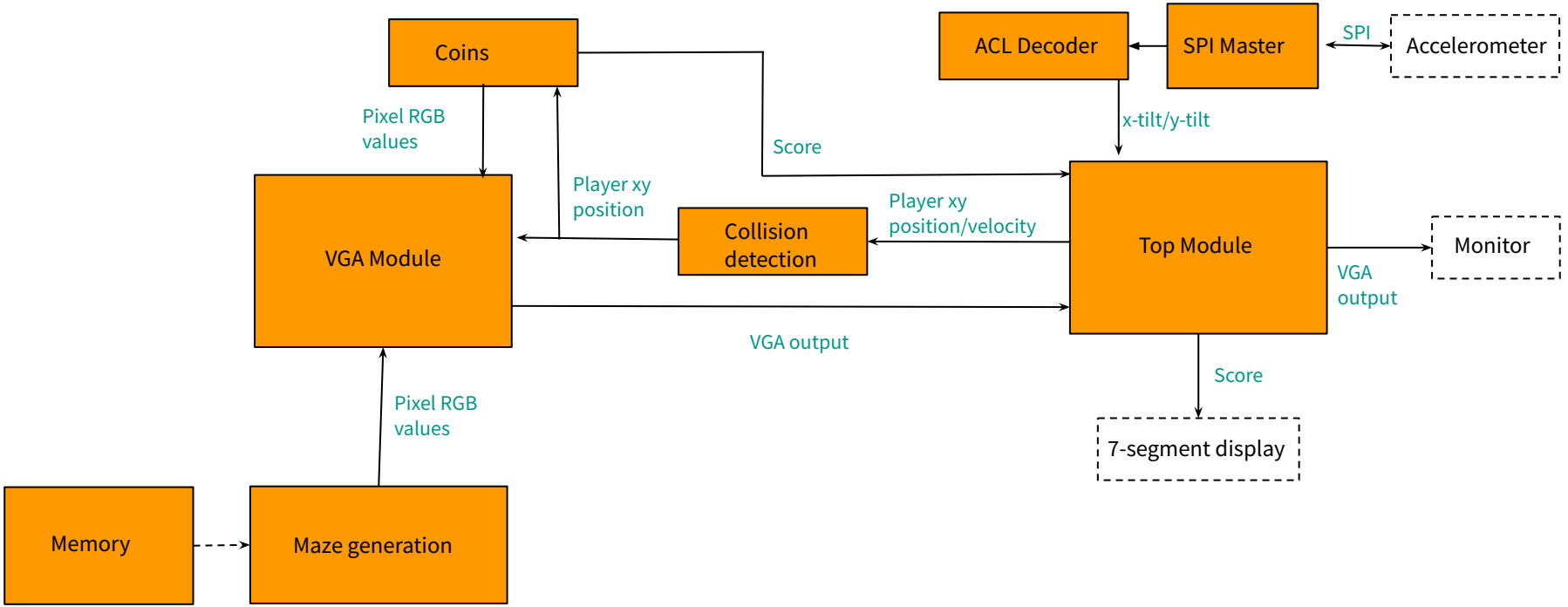
Goal/Motivation/Specs

- Goal
  - A simple game using the accelerometer for motion control and displayed on a screen
- Game Specs
  - Maze displayed on screen with player trying to get to the top
  - Movement of player determined by orientation of board
  - Player collides with maze walls and coins
  - Score display on 7-seg display keeps tracks of current score
  - Multiple maps
- Motivation
  - Wii sports





Detailed Block Diagram

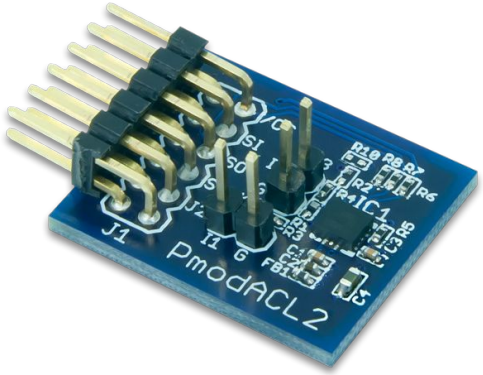


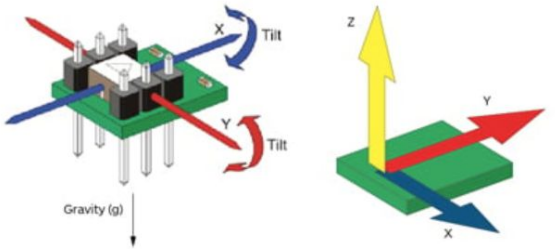
Accelerometer (ADXL362)

We are going to use the ADXL362 to control the top-down movement of the ball. This accelerometer will send us information about how the FPGA is tilting, allowing us to use that information for a control scheme.

Using this we will read information from the y-axis and z-axis data registers and translate that into usable data for moving the ball on screen.

We will have to use a State Machine for the SPI Interface in order to handle the accelerometer and ensure it is working properly





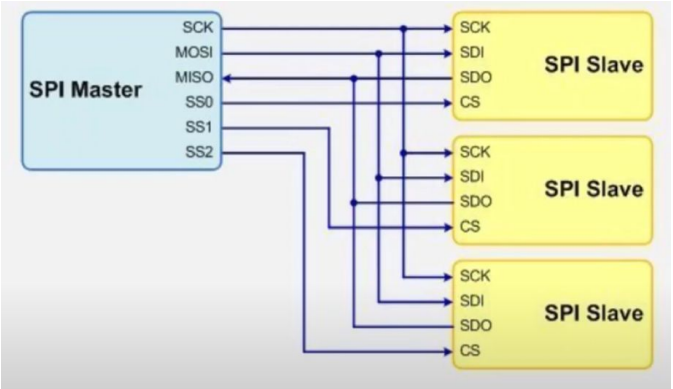
SPI Interface and Master

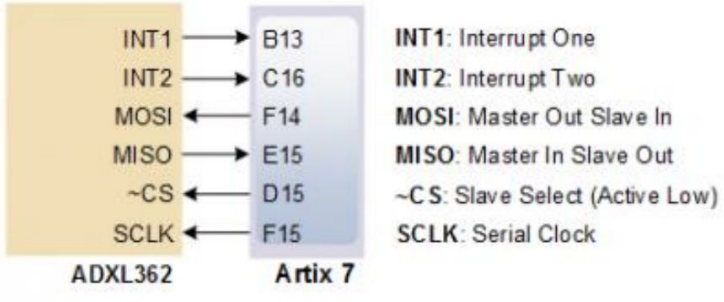
The SPI Interface and Master modules are what control and tell the Accelerometer what to do. Using the SPI interface, we can send specific instructions to the accelerometer to read information from it's sensors.

Since we only have one control, the SPI Master will not have to change which SPI Slave it is interacting with. Thus, we can focus on sending the correct instructions.

By sending a series of bits as instructions to the accelerometer, we can read the data for the tilting from their respective registers. While the SS/CS signal is low, and at the posedge of the Serial Clock, it will detect different bits, allowing us to do that.

However, the accelerometer works at a much lower clock frequency, at around 4MHz, so we will need a clock divider





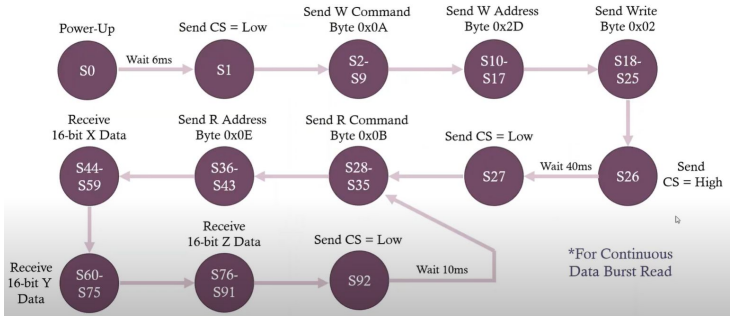
Accelerometer State Machine and Data

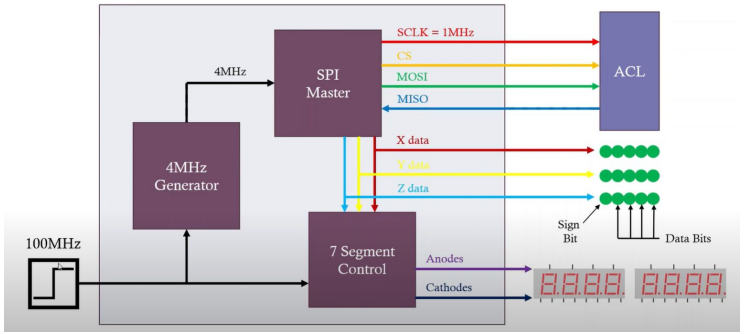
Since the accelerometer follows very specific instructions using different states, we'll need a FSM.

By creating this state machine, it'll allow us to read from the accelerometer in order to accurately enable movement.

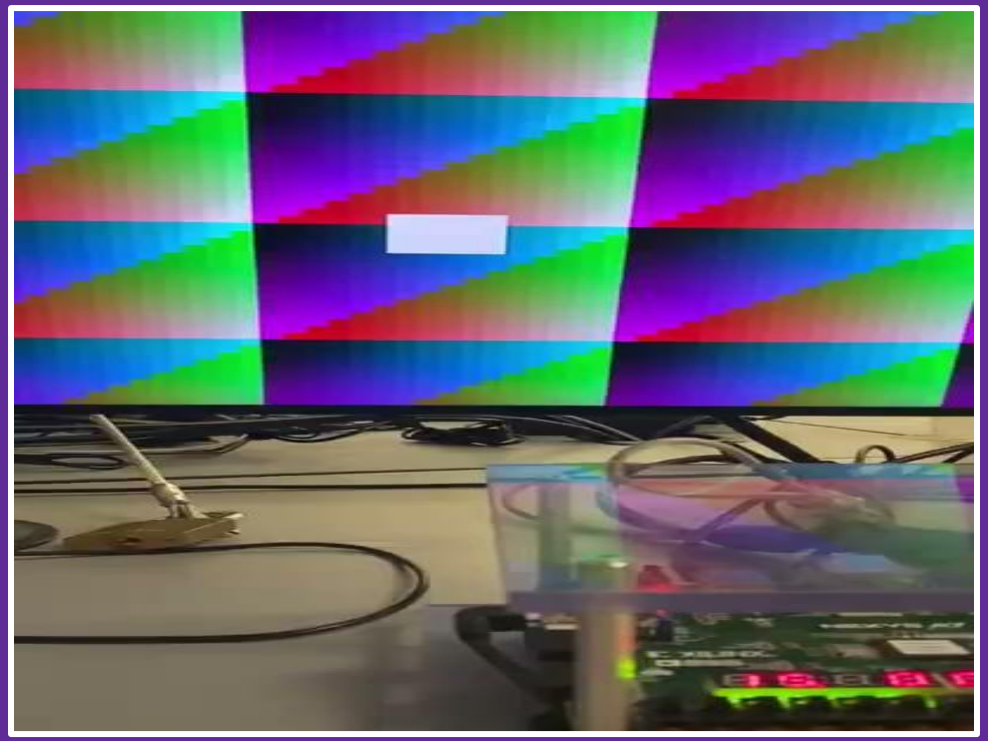
We will need to take the sign bit, as well as some bits to represent magnitude in order to represent the direction and the degree of tilting.

The setup of the accelerometer is very particular, since we'll need to put into measurement mode, that's what the first 7 states of the FSM are for. Combining the FSM with the SPI Master, we'll be able to use the accelerometer data for motion controls.





Accelerometer Example



Accelerometer Attempt Timeline

- Original Attempt (2 Hours):
  - Looking directly at Datasheet
  - Creating custom clocks and signals
  - Creating opcode parameters for read/write
  - Flaw: ACL never went into “measurement mode”
- Second Attempt (1.5 Hours):
  - Used MIT Pong Github Repo for Help using Spart 6
  - Converting functions to work for Nexys A7
  - Making an SPI interface since there’s was too different
  - Flaw: Accelerometers were too different and difficult to port over
- Final Attempt (2 Hours):
  - Used guide that the Guest lecturers used on interfacing with the ACL
  - Utilized the burst read opcode rather than register read
  - It was for the Nexys A7, so it was completely compatible
  - Success!

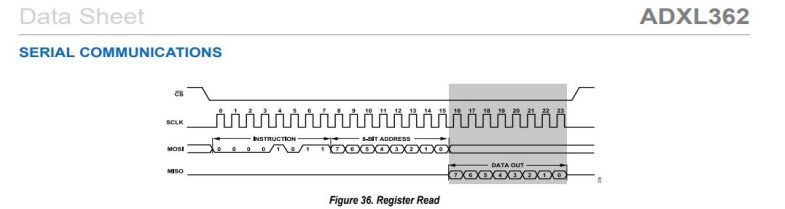
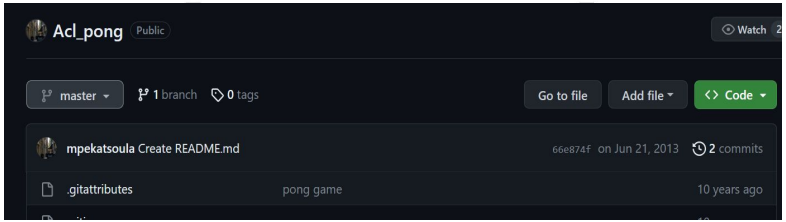
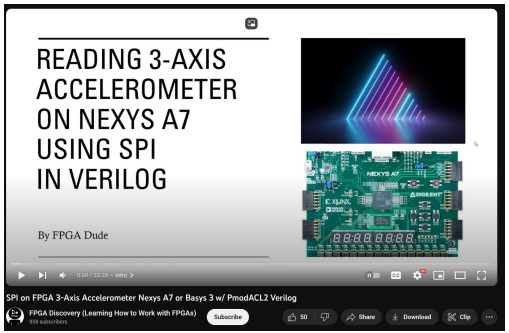


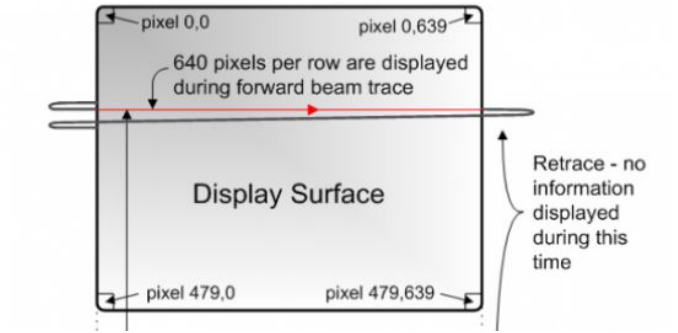
Figure 36. Register Read

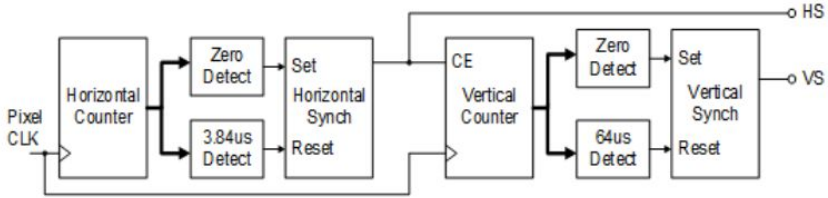




VGA Controller

- Controller displays one pixel at a time
  - Set by a horizontal and vertical counter
  - Vertical counter increments once for every time that the horizontal counter resets.
  - Specific timings (3.84 us and 64 us) set by the resolution of the display and refresh rate



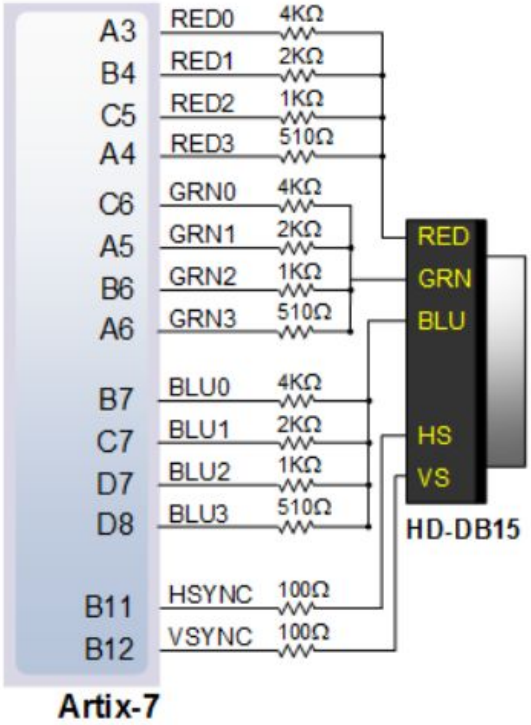


https://digilent.com/reference/programmable-logic/nexys-a7/refere-manual

VGA Controller

- Twelve bits per pixel
  - (4 bits for each color of RGB).
  - Could be loaded from memory.
    - Vertical and horizontal counters used as address.
- Currently, the pixels are manually set within geometries.
  - Code shows pixels in a rectangle being set to a certain color.
- Positions updated according to the orientation of the accelerometer for movement
  - Previous attempts shredded image

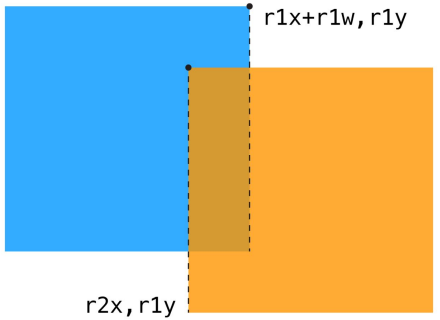
```
//door
if ((vga_hcnt >= (0) && vga_hcnt <= (75)) &&
(vga_vcnt >= (225) && vga_vcnt <= (275))) begin
VGA_RB = 7;
VGA_GB = 3;
VGA_BB = 0;
end
```



https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual

Collision Detection

The player's movement is determined by the accelerometer and displayed by the VGA. The next step is to **prevent movement** when the player collides with one of the walls of the maze. This can be done using 2D **rectangle-to-rectangle collision detection.** The ball in the maze will have a square "hitbox", and code will detect whether this hitbox is about to collide with one of the rectangular walls of the maze. By looking at the position of the ball before and after the collision, it can be determined which edge of the ball will collide with the wall and therefore which direction to prevent movement. Alternatively, the velocity of the ball could be reversed instead of set to zero upon a collision to cause the ball to "bounce" off the wall instead of sliding.



https://www.jeffreythompson.org/collision-detection/rect-rect.php

Collision Detection

```
23  module movement(
24  input clk, reset, //clk will probably be the refresh rate of the screen, 60 Hz
25  input [31:0] vx,vy, //vx and vy are the x and y velocity (get from x-tilt and y-tilt of accelerometer)
26  input [31:0] x1,y1, //current x and y position
27  input [31:0] wall_left, wall_right, wall_bottom, wall_top,
28  output reg [31:0] x_out, y_out //new x and y position
29      );
30
31      //determine new position (assuming no collisions)
32      wire x2,y2;
33
34      assign x2 = x1 + vx;
35      assign y2 = y1 + vy;
36
37      //determine positions of square edges before and after collision
38      parameter square_length = 50;
39      wire [31:0] square_top1, square_bottom1, square_left1, square_right1;
40      wire [31:0] square_top2, square_bottom2, square_left2, square_right2;
41
42      assign square_left1 = x1;
43      assign square_right1 = x1 + square_length;
44
45      assign square_bottom1 = y1;
46      assign square_top1 = y1 + square_length;
47
48      assign square_left2 = x2;
49      assign square_right2 = x2 + square_length;
50
51      assign square_bottom2 = y2;
52      assign square_top2 = y2 + square_length;
```

Collision Detection

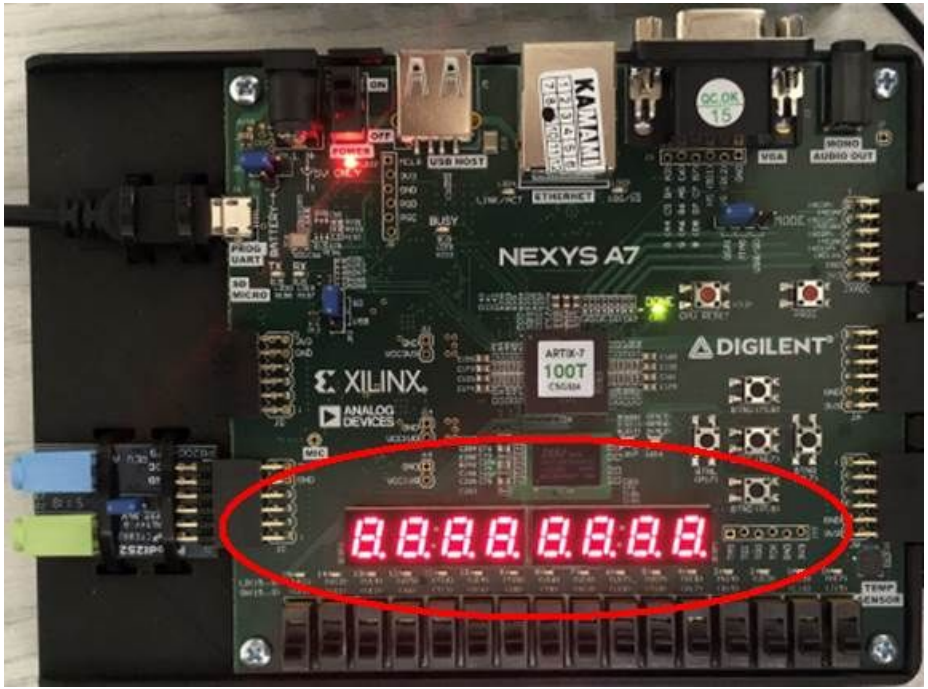
```
54      //flags for collisions
55      reg collision, collision_left, collision_right, collision_bottom, collision_top;
56
57      //check if there will be a collision, then determine which side the collision happened from
58      always @ * begin
59      if (square_left2 <= wall_right && square_right2 >= wall_left && square_bottom2 <= wall_top && square_top2 >= wall_bottom) begin
60      collision = 1;
61          if (square_left2 <= wall_right && square_left1 > wall_right)
62          collision_left = 1;
63          else
64          collision_left = 0;
65          if (square_right2 >= wall_left && square_right1 < wall_left)
66          collision_right = 1;
67          else
68          collision_right = 0;
69          if (square_bottom2 <= wall_top && square_bottom1 > wall_top)
70          collision_bottom = 1;
71          else
72          collision_bottom = 0;
73          if (square_top2 >= wall_bottom && square_bottom1 < wall_top)
74          collision_top = 1;
75          else
76          collision_bottom = 0;
77      end
78      else
79      collision = 0;
80      end
81
82      //output new position, preventing movement in collision direction if collision occurs
83      always @ (posedge clk) begin
84      if (collision_left == 1 || collision_right == 1)
85      x_out = x1;
86      else
87      x_out = x1 + vx;
88      end
89
90          always @ (posedge clk) begin
91          if (collision_bottom == 1 || collision_top == 1)
92          y_out = y1;
93          else
94          y_out = y1 + vy;
95          end
96
97  endmodule
98
```

Seven Segment Display

The Seven Segment Display will be used to keep track of the players current score.

Players can earn points in three situations:

1. When the game detects the player has reached the end.
2. When the player collides (picks up) a coin.
3. When the game detects the player has picked up all coins on the map (bonus points).



Maze Generation

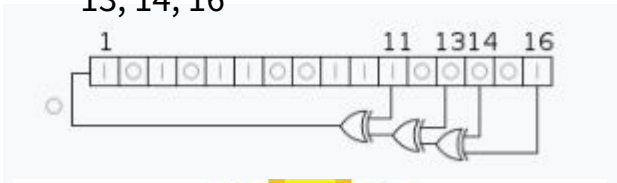
Using LSFR (Linear-feedback shift register), levels could be generated with random blocks.

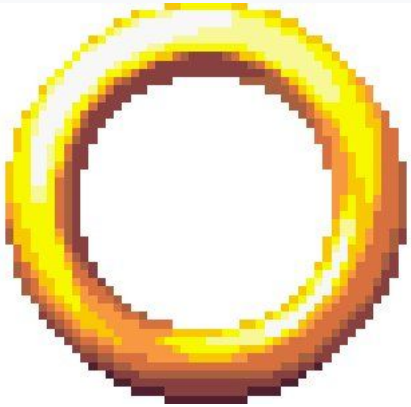
Otherwise, a set of levels could be stored in memory.

Once the player reaches the end, they are reset to the bottom of the screen and a new level is displayed/generated.

Coins will be placed randomly in the maze where if collected, will provide bonus points for the player.

16-bit LSFR with taps on bits 11, 13, 14, 16





Thank you

Github: https://github.com/cj-casey/EC311-Project