Final Project

Pac-Man on an FPGA

Tufts University, Fall 2021

ES 0004: Introduction to Digital Logic

Alec Plano, Evan Loconto,
Hannah Magoon, Zachary Goldstein

Contents

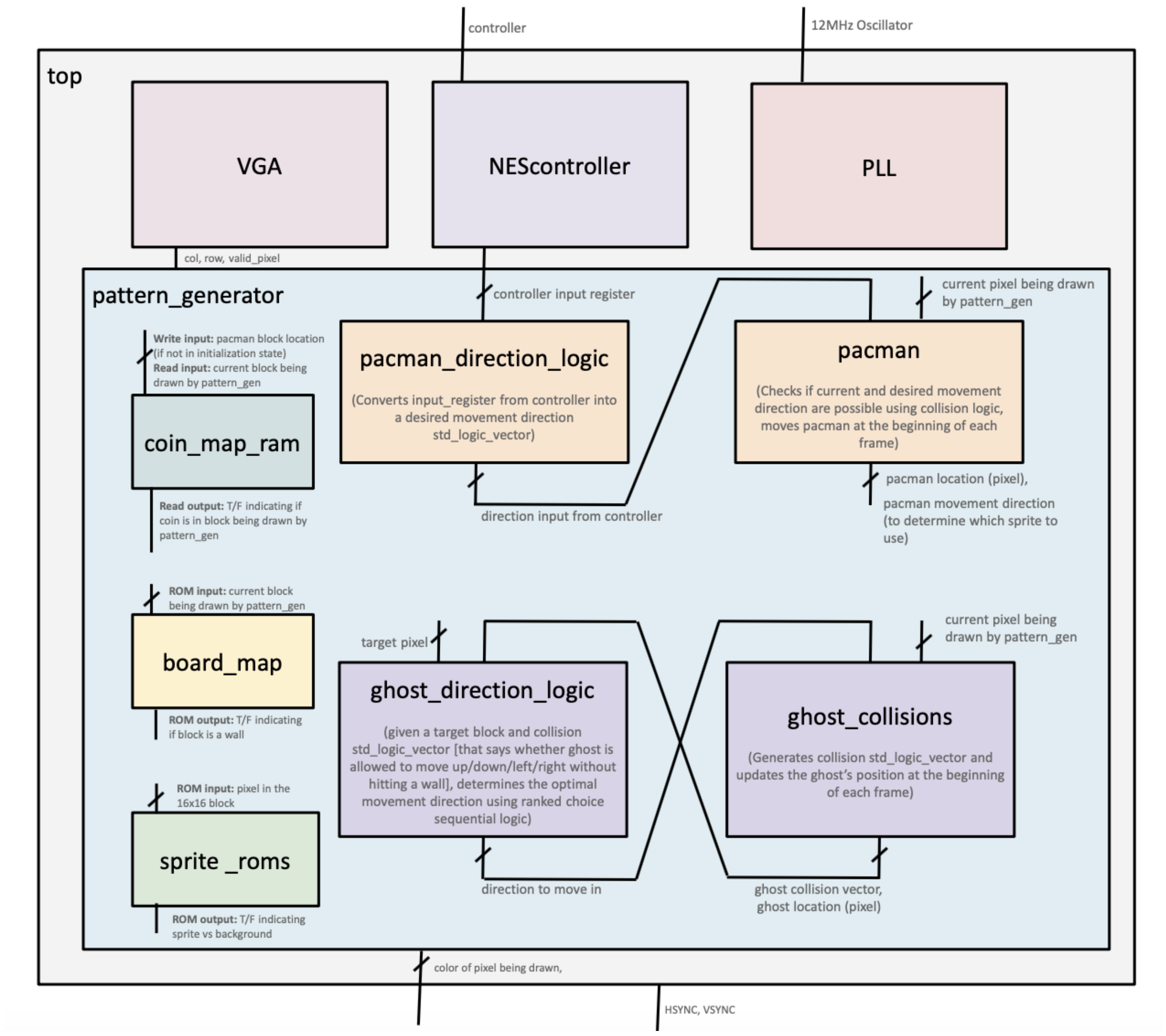
1 Overview

To begin this project, we divided the screen into a 40x30 grid of boxes. Each of these boxes is a 16x16 pixel square. Throughout this report, we will use the terms "block" and "pixel" location. "Block" location refers to a block number on this grid. "Pixel" location refers to the row and column location of a specific pixel. We stored the locations of Pac-Man and the ghosts as pixels so that their motion across the board could be fluid, but all other information is stored as a block to save space.

A rough sketch of the block diagram is included on the next page, but only the important inputs/outputs are included to keep the image somewhat readable. Although not drawn on the diagram, all entities will receive a clock that changes as each new pixel and/or frame is drawn, and the non-memory entities will receive a reset signal.

As shown in the diagram, most entities are nested inside the pattern generator. A board map ROM is used as a look up table. Given a block address, its output will indicate whether the block is a wall or a path. A RAM is used to store whether a coin is located at the given pixel. As Pac-Man moves, '0' is written into the RAM at his current block location, meaning that the coin at his current location is removed. Because the RAM is dual ported, the read input/output can be used to simultaneously determine if a coin is in the block that the pattern generator is currently drawing.

Pac-Man and the ghosts are each controlled by two entities. One of them sets the desired movement direction, and the other moves the sprite once per frame. Their sprites are stored in 16x16 pixel ROMS, which override the board map pixel color when the pixel being drawn falls within the bounds of one of these sprites. The game is lost when Pac-Man's sprite occupies the same location as one of the ghosts. More details on our implementation are provided in the next section, but hopefully this overview makes the following block diagram somewhat intelligible!


controller
12MHz Oscillator
top
VGA
NEScontroller
PLL
col, row, valid_pixel
pattern_generator
controller input register
current pixel being drawn by pattern_gen
Write input: pacman block location (if not in initialization state)
Read input: current block being drawn by pattern_gen
coin_map_ram
Read output: T/F indicating if coin is in block being drawn by pattern_gen
pacman_direction_logic
(Converts input_register from controller into a desired movement direction std_logic_vector)
direction input from controller
pacman
(Checks if current and desired movement direction are possible using collision logic, moves pacman at the beginning of each frame)
pacman location (pixel),
pacman movement direction (to determine which sprite to use)
ROM input: current block being drawn by pattern_gen
board_map
ROM output: T/F indicating if block is a wall
ROM input: pixel in the 16x16 block
sprite _roms
ROM output: T/F indicating sprite vs background
target pixel
ghost_direction_logic
(given a target block and collision std_logic_vector [that says whether ghost is allowed to move up/down/left/right without hitting a wall], determines the optimal movement direction using ranked choice sequential logic)
direction to move in
current pixel being drawn by pattern_gen
ghost_collisions
(Generates collision std_logic_vector and updates the ghost's position at the beginning of each frame)
ghost collision vector, ghost location (pixel)
color of pixel being drawn,
HSYNC, VSYNC

2 Technical Description

2.1 Programming Conventions

To maximize the readability and debug-ability of our codebase, we adopted the following conventions early in development:

1. component input signals are appended with suffix "_i"
2. component output signals are appended with suffix "_o"
3. internal component signals do not have a suffix
4. std_logic vectors are preferred over numeric types
5. general purpose components are preferred over case-specific components to maximize code reuse
6. with rare exception, all non-blocking assignments must occur within process blocks to ensure consistent signal propagation
7. the top-level module serves primarily to instantiate instances of sub-components. Minimal logic is implemented directly within the top-level module
8. the PLL oscillator used to drive the VGA clock is used as the clock for every other component. This ensures that all components and signals are updated synchronously.

2.2 VGA Display

The VGA display is composed of three parts, the first is the PLL which is in charge of creating a very accurate clock signal. This clock signal is then used by the VGA module which counts each pixel across the screen and includes resting time at the right and bottom of the screen. This module outputs the vertical and horizontal sync signals to the VGA port and the pixel X and Y position to the pattern generator. The final piece of the VGA display is the pattern generator, which takes the X and Y pixel position and calculates which color that pixel should be. This color data is then sent to the VGA port.

2.3 NES Controller

A NES controller is used to relay user input to the pattern generator. Because the clock frequency required by the controller is slower than that of the PLL, a counter was used to convert the pixel clock signal to a NES clock within the specified range. With this clock signal, the NES entity was tasked with sending and receiving information from the controller. Here, a latch signal was needed to set the timing for the button inputs. When a button is pressed on the controller, a signal is sent on a specified clock cycle after the latch input.

This signal is read into the entity and translated to an eight-bit shift register. This information is fed into the Pac-Man movement direction logic entity.

2.4 Board Map and Sprite ROM

The sprite and board map ROMs were built from large lookup tables. The board map ROM takes in a block number and outputs a '1' if the block is a wall, and a '0' if it is a path.

The sprite ROMs take in a (pixel) address within the 16x16 pixel block, and outputs a '1' if the pixel is part of the sprite image, and a '0' if it is part of the background. In order to make these lookup tables we manually wrote out lists of 1's and 0's. Then we used python to convert our array of bits into VHDL syntax. From here, we created instances of the ROMs inside the pattern generator to provide information about the current pixel being drawn.

2.5 Coin Map RAM

The RAM works similarly to the ROM: as the screen is being generated we read from RAM and draw coins where it is necessary. On initialization of the game, every spot in the board is given a coin, but the ones that are stored in wall spaces are removed after the first frame is drawn. This leaves us with coins in the open spaces only. Then as Pac-Man moves we feed his block location into the write address port of the RAM, and write over the value stored in the memory with a '0'. Regardless of its previous value, the block is converted to a non-coin location. The code to initialize and write into the RAM is as follows:

```vhdl
signal coin_init : std_logic := '0';
---
if (rising_edge(clk_i)) then
  if(coin_init = '0') then --initialize on first frame
    --note: pixel_curr_block is the block # of the pixel being
        drawn, and rom_out is the output of the board map ROM
    w_addr <= std_logic_vector(pixel_curr_block);
    w_enable <= '1';
    w_data <= '1' when rom_out = '0' else '0';
    if(pixel_curr_block = "10010101111") then
      coin_init <= '1';
    end if;
  elsif(input_register_i(4)) then --reset signal
    coin_init <= '0';
  else
    r_addr <= std_logic_vector(pixel_curr_block);
    w_enable <= '1';
    w_addr <= std_logic_vector(pacman_curr_block);
    w_data <= '0';
  end if;
end if;
```

2.6 Pac-Man

The Pac-Man sprite is controlled by two separate VHDL entities. The first one, Pac-Man direction logic, takes in an input register from the NES controller and turns it into a 2-bit direction instruction. This instruction is wired through a signal in the pattern generator into the Pac-Man entity. Here, Pac-Man's current movement direction and the desired movement direction (from the controller input) are tested using collision logic. At the beginning of each frame:

– If movement in the desired-movement direction is possible, Pac-Man is moved and the desired-movement direction is set as the current-direction.

– Otherwise, if movement in the current-movement direction is possible, Pac-Man's location is moved by one pixel in that direction.

2.7 Collision Logic

The collision algorithm is implemented as an inter-operable component that is instantiated for each moving game entity. This component takes as input: the current location of the relevant entity, the current pixel being drawn to the screen, and whether that drawn pixel is a wall pixel. The logic is implemented with a set of 8 test pixels that lie immediately adjacent to key locations on the entity bounding box. If one of these pixel is the color of a wall, then that portion of the entity bounding box is adjacent to a wall.

A std_logic vector (7 downto 0) contains these 8 adjacency signals. The location of each test pixel is determined by the location of the relevant entity as summarized below:

0. top left	(xpos, ypos - 1)
1. top right	(xpos + 15, ypos - 1)
2. bottom right	(xpos + 15, ypos + 16)
3. bottom left	(xpos, ypos + 16)
4. left bottom	(xpos - 1, ypos + 15)
5. left top	(xpos - 1, ypos)
6. right top	(xpos + 16, ypos)
7. right bottom	(xpos + 16, ypos + 51)

In the figure below, the red pixels represent the adjacency markers, the yellow pixels represent an entity (Pac-Man), the blue pixels represent walls, and the black pixels represent pathways.

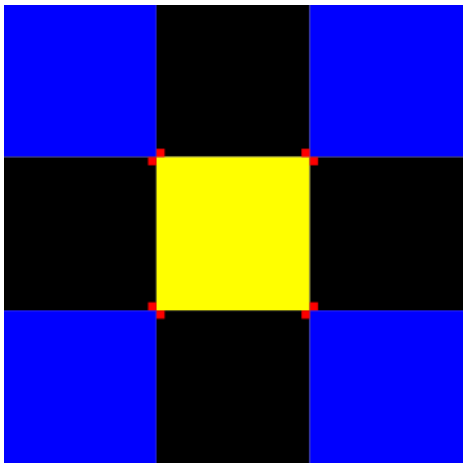


Figure 2.1: The relative location of the 8 adjacency markers. In this example, Pac-Man can move in any cardinal direction.

These signals are condensed into a std_logic vector of four signals, can_move_o, where each signal represents whether the entity can move in that direction:

0. can move **up** *iff* neither of the above adjacency markers are triggered
1. can move **down** *iff* neither of the below adjacency markers are triggered
2. can move **left** *iff* neither of the left adjacency markers are triggered
3. can move **right** *iff* neither of the right adjacency markers are triggered

To further demonstrate the implementation of this collision detection algorithm, the next three figures illustrate other scenarios.

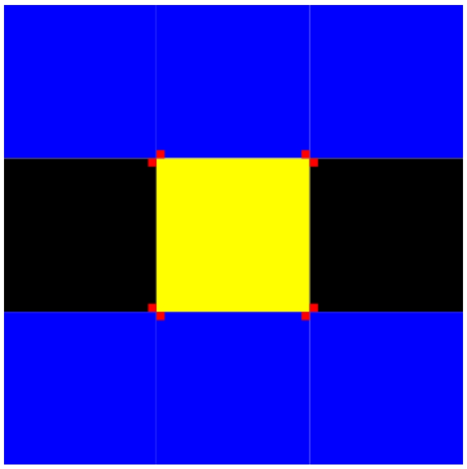


Figure 2.2: Pac-Man can move left or right

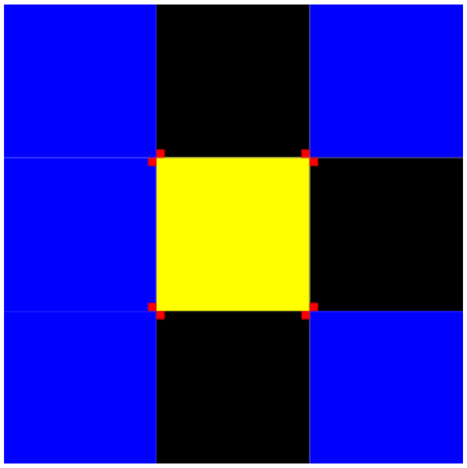


Figure 2.3: Pac-Man can move up, down, or left

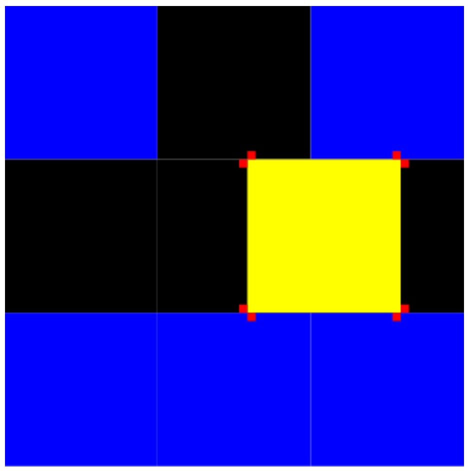


Figure 2.4: Pac-Man can move left or right, but *not* up. The top right buffer is triggered, preventing Pac-Man from moving into the wall.

The code is provided below to resolve any residual confusion:

```
if (rising_edge(clk_i)) then
        collisions(0) <= is_pixel_wall_i when (col_i = (xpos_o )
            and row_i = (ypos_o - 1)) else collisions(0); -- top
            left
        collisions(1) <= is_pixel_wall_i when (col_i = (xpos_o +
            15) and row_i = (ypos_o - 1)) else collisions(1); --
            top right
        collisions(2) <= is_pixel_wall_i when (col_i = (xpos_o +
            15) and row_i = (ypos_o + 16)) else collisions(2);
            -- bottom right
        collisions(3) <= is_pixel_wall_i when (col_i = (xpos_o )
            and row_i = (ypos_o + 16)) else collisions(3); --
            bottom left
        collisions(4) <= is_pixel_wall_i when (col_i = (xpos_o -
            1) and row_i = (ypos_o + 15)) else collisions(4); --
            left bottom
        collisions(5) <= is_pixel_wall_i when (col_i = (xpos_o -
            1) and row_i = (ypos_o )) else collisions(5); --
            left top
        collisions(6) <= is_pixel_wall_i when (col_i = (xpos_o +
            16) and row_i = (ypos_o )) else collisions(6); --
            right top
        collisions(7) <= is_pixel_wall_i when (col_i = (xpos_o +
            16) and row_i = (ypos_o + 15)) else collisions(7);
            -- right bottom
     can_move_o <= (collisions(7) nor collisions(6)) & --right
                  (collisions(5) nor collisions(4)) & -- left
                  (collisions(3) nor collisions(2)) & -- bottom
                   (collisions(1) nor collisions(0)); -- top

      end if;
```

2.8 Ghosts

The ghost sprites are controlled using a similar setup to Pac-Man.

2.8.1 Ghost AI

Although referring to the ghost direction logic as "artificial intelligence" is quite generous, it nonetheless is the case that the core ghost AI always makes the optimal direction decision at every intersection.

The ghost direction logic entity takes in the ghost's current location (pixel), the ghost's target location (pixel), and a 4-bit collision vector that indicates whether the ghost can move up/down/left/right. The ghost AI may choose the current location of Pac-Man as its target location, or some other "home-base" location that is provided during instantiation.

Note: The ghosts cannot reverse their direction. At any intersection, a ghost may choose to continue in its current direction, turn left, or turn right. A ghost will never leave an intersection from the same path that it used to enter. This results in consistent and relatively bug-free ghost movement. Critically, this prevents ghosts from getting stuck.

The direction entity compares the horizontal and vertical distances between the ghost's current location and its target location, and chooses a direction that minimizes the larger of the two if possible. It determines whether it prefers horizontal or vertical motion, and whether it prefers up over down and left over right. Then, using a long chain of combinational logic, it determines which of the allowed movement directions is optimal. There are thirty-two unique configurations of intersections and preferred movement directions, so the logic is implemented with nested conditionals.

2.8.2 Clock-Cycle Confusion

Although difficult to describe in a report, this logic implementation was relatively straightforward. The only issue is that it takes an entire frame to determine the collisions around any particular sprite location. This is because the collision logic we implemented does not have its own instantiation of the board map ROM. Instead, it pulls from the one being used to draw the screen. To determine whether a given pixel is a wall or a path, the collision detector waits for the pattern generator to draw the pixel and stores its value. This means that by the time that the collisions around a given sprite location are calculated, the frame has been drawn and the sprite needs to be moved. This left us with an off-by-one clock cycle error.

At first, we tried to resolve this using the dead time between frames, but introducing new clocks complicated things, and the ghosts did not always follow the shortest path to Pac-Man. To resolve this issue, we calculated collisions in the 4 blocks around the ghost's current location. Then, when the ghost's location was updated, the correct collision logic vector could be selected, giving the movement logic entity an entire frame to calculate the optimal ghost direction. This resolved the issue, and the ghosts were able to effectively move their target locations.

2.8.3 Pursuit and Scatter

To prevent ghosts from lining up in a row and following Pac-Man, we alternated between setting their target location to Pac-Man and to one of the four corners on the board. At any given time, two ghosts are sent towards Pac-Man, and the other two are sent to their home corners. A ghost is in "pusuit" mode when its target position is Pac-Man's position, and "scatter" mode when its target position is its home-base corner. This is very similar to the ghost AI behavior in the real Pac-Man game!

3 Results

3.1 VHDL Code

https://github.com/Evanloconto/Pacman-FPGA

3.2 Project Demo

https://youtu.be/r30P7GXCb5I

3.3 Remaining Bugs

3.3.1 Win Condition

In its current state, the game is unable to determine when all of the coins have been eaten. This means that although a player can lose the game, they can never truly win. Implementing a "win" condition should be fairly straightforward. To do so, we could create a signal in the pattern generator and set it to '0' at the start of each frame. If a coin is drawn to the board, the signal would be flipped to a '1'. If the signal is still '0' at the end of the frame, then there are no coins left on the board. This means that Pac-Man has won the game.

We faced a number of bugs while implementing this, and the game would be 'won' at seemingly random times during play. Even though the concept described above is fairly simple, implementation was complicated by the fact that one signal cannot be driven by two different process blocks. Because we ran out of time to figure out the source of these bugs, the feature had been commented out of the final version of the code.

3.3.2 Tunnel Glitches

We still have two small bugs associated with the tunneling feature of the game. The first is that when Pac-Man goes through the tunnel from the left side of the board to the right, he misses the right-most coin in the tunnel. However when he enters the tunnel from the right side of the board, he gathers all of the coins as expected.

The other bug associated with the tunnel has to do with the ghosts. Despite all the ghosts logic being identical (with only small variations on target location), two of the ghosts have a tendency to glitch inside the tunnel. By this we mean that the ghost will be going towards the tunnel, but right before crossing to the other side of the board, the ghost's direction reverses by 180 degrees. Because ghosts should never completely reverse direction, this may due to a clock cycle error introduced when we hard-coded teleportation through the tunnel.

These two bugs are the most straightforward to fix, but since they have almost no impact on game-play, they are low on our priority list.

3.3.3 Orange Ghost Movement Glitch

While coding and testing this game, we noticed that the orange ghost occasionally gets stuck in corners. We observed this happening on four separate occasions. Because this error happened so infrequently (and could not be repeated, even when we had Pac-Man follow the same path), we are unsure of why this is occurring. It is interesting to note that in the pattern generator entity, the orange ghost's combinational logic is listed last.

The location in the ghost's movement logic that corresponds to the decision being made at the time of one of these glitches was marked with a comment, but there do not appear to be any problems with the code. The fact that the ghost is able to make the correct movement direction the vast majority of the time leads us to believe that this error is not due to faulty logic. We are unsure how to debug this, and believe that it may be a sequential timing error? Introducing additional Pac-Man lives makes this issue worse (it was only observed once without the additional sequential logic). It's also worthwhile noting that while compiling the code, we get a warning about clock timing.

3.4 Things we ran out of time for

Given more time on this project, we would:

- Clean up the code and add comments to make it readable
- Attempt to resolve the bugs described above
- In real Pac-Man, each ghost is sent towards a different target location. Given more time, we would read about the ghost behavior and write some combinational logic in the pattern generator to give the ghosts different objectives
- Create board ROMS to display when someone wins/loses the game (instead of coloring the screen green/red)
- Animate the pacman sprite to make it look like he's eating the coins :-)
- Add cherries to the game. Although this sounds difficult, it wouldn't require too much new code/logic. We would create another RAM, store the ghost's state in a signal, and simply change the ghost target position (and maybe speed) to have them run away from pacman. We would also have to adjust the end of game logic slightly (so that pacman could eat a ghost without dying), but this would be relatively straightforward
- We wanted to have the ghosts start in the center square at the beginning of the game, and move out after a set number of clock cycles. This would be a bit tricky to implement given the structure of our code, but we could

fake it by drawing a ghost sprite in the center box, and waiting a given interval of time before placing the real ghost entity on the map

4 Reflection

This project went fairly smoothly. Our group worked well together, and although we got stuck at a couple of points, we were able to resolve the majority of our bugs. The pace that we worked on this project was a bit erratic, but it actually helped us to avoid the stressful time crunch that some of our friends had to deal with. Although it would have been nice to have more time to work on this, it probably would have been at the expense of our other finals. To put things simply, we genuinely had fun with this project and we would not change much if we were to do it over again. That being said, if I knew we were going to be in the lab so late I would have ordered pizza.

5 Work Distribution

It's difficult to describe individual contributions to the project because we met up and worked together rather than breaking the project into small subtasks. Furthermore, we had to continuously modify existing code to mesh with new additions to the project. This means that entities were passed around and rewritten multiple times. Nevertheless, a rough description of our contributions is outlined below.

Contribution	**Team Members**
Circuit	Alec, Zach, Evan, Hannah
VGA Display	Alec, Zach
NES Controller	Evan, Zach
Board Map ROM	Alec, Hannah
Sprite ROMs	Alec, Evan
Coin Map RAM	Alec, Hannah
Collision Logic	Zach, Evan, Hannah
Pac-Man	Zach
Ghost AI	Zach, Hannah
Ghost Clock-Syncing	Hannah

6 Acknowledgements

We would like to give a quick shoutout to all of our TAs for their help and support (especially John, who explained a better way to implement collision logic and suggested that we learn to use variables). We would also like to thank for Professor Bell for taking the time to talk through the design of this project, helping us debug, and for telling us about the secret lab room down the hall.