Pac-Man EXTREME!!!!!™

Kim Dauber and Rachael Devlin

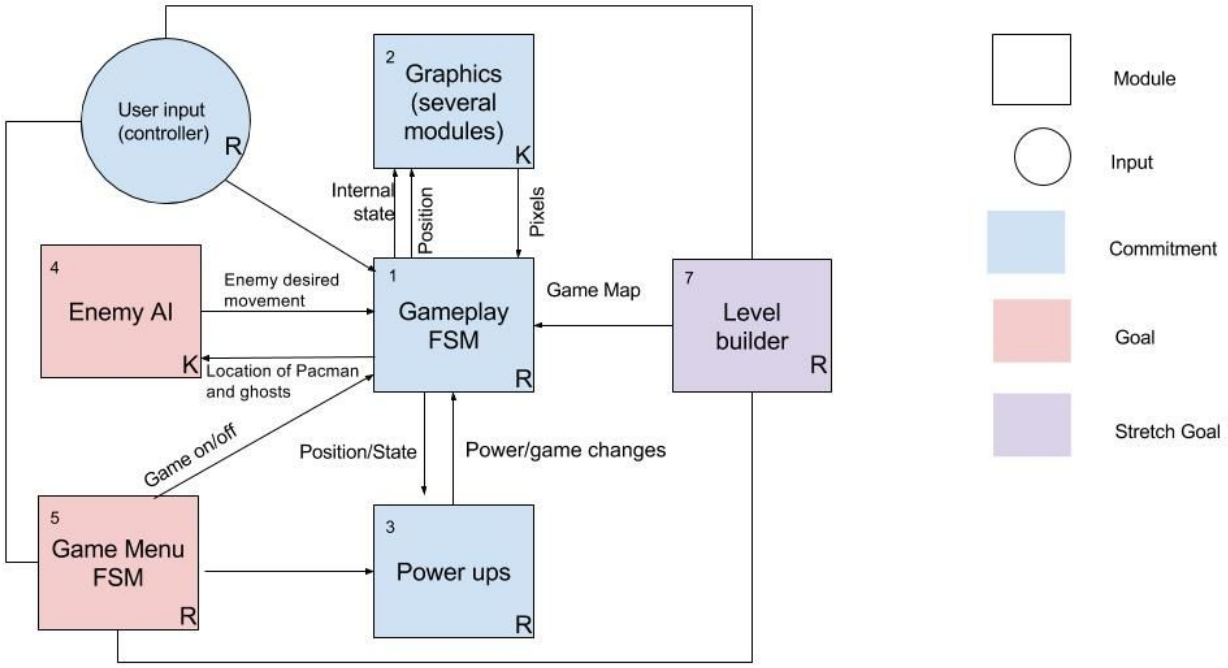
6.111 Fall 2017

Introduction

Is regular old Pac-Man too boring for you? Your boring days of dot-gobbling are over, because here comes Pac-Man EXTREME!!!!!™

Pac-Man EXTREME!!!!!™ is two-player Pac-Man with powerups and a level editor. Grab powerups to destroy your competitors' chances at victory. Use your controller to create a level of Pac-Man. You draw the walls and the FPGA makes your vision come to life, ready to play.

Block Diagram



Modules

Graphics - Kim

The graphics modules generate pixel outputs that can display game graphics on the screen through VGA. Most sprites are 16 x 16 pixels and uses 12-bit color. (The letter sprites used on the game menu are 8 x 8.) The graphics modules vary in complexity. The Pac-Man and ghost modules are by far the most complicated because Pac-Man requires the most animation and the ghosts have the most potential states in the game. Next in complexity come the food dots and large dot power-ups, which use alpha blending to make smooth curves. Finally, the sprites for the other powerups, letters, and numbers are quite simple since they do not involve any animation or curved lines. Our main source for what Pac-Man sprites actually look like came from Simon Owen's Pac-Man emulator writeup.[1]

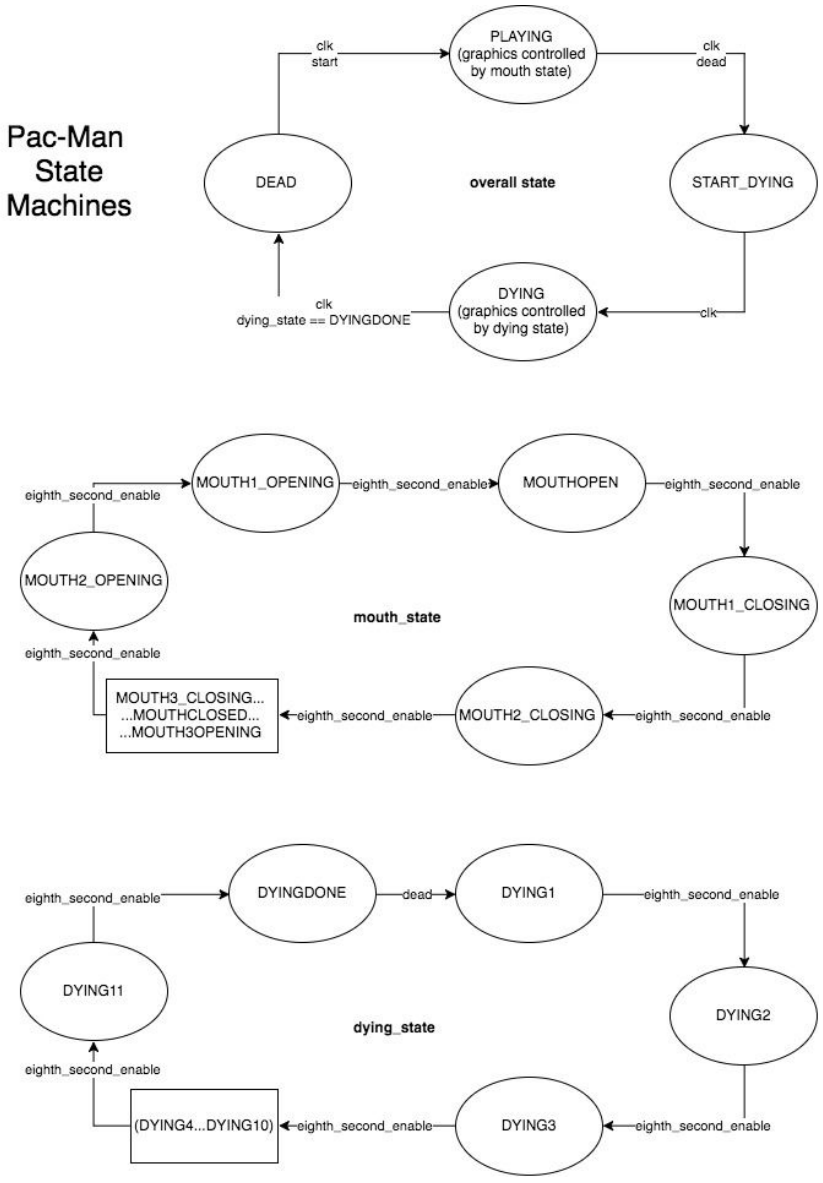
These are the inputs and outputs of a typical simple graphics module.

Port type	Name	Description
input	`clk`	25 MHz clock signal
input	`reset`	reset signal
input	`hcount[10:0]`	horizontal coordinate of pixel
input	`vcount[9:0]`	vertical coordinate of pixel
input	`x[10:0]`	x position of the sprite's top left corner
input	`y[9:0]`	y position of the sprite's top left corner
output	`pixel[11:0]`	pixel color

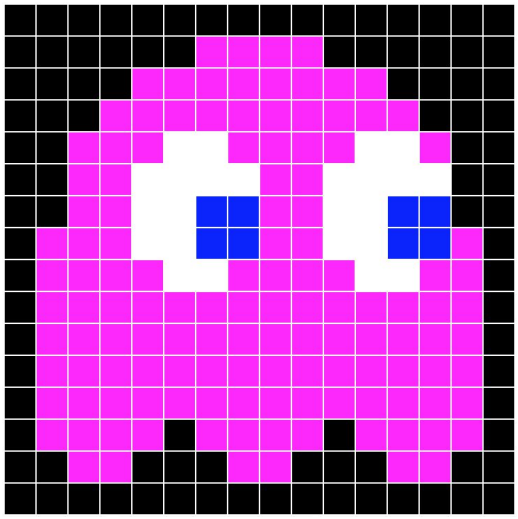
Each individual sprite is a series of 256 if/else statements, one for each pixel in the sprite. Sprites with curves also have an input called `inner_graphics` and alternate between two sets of pixels, one for the pixels on the inside of the curve and one for the outside. This creates an alpha blending effect for the appearance of a smooth curve. Over two hundred fifty if statements per sprite would be a lot of verilog to write and debug, so we built a sprite maker app that provides a graphical interface to create a sprite and generate the corresponding verilog. This is also the reason that all the sprites' code is the same format. The app can currently (as of December 2017) be found at http://kadauber.scripts.mit.edu/sprite-maker/. Its source code can also be found on Github at https://github.com/kadauber/sprite-maker and https://github.com/kadauber/sprite-maker-api.

[1] http://simonowen.com/sam/articles/pacemu/

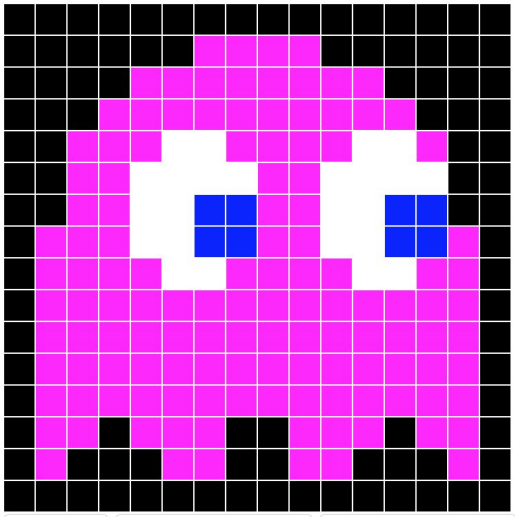
The Pac-Man graphics are relatively complicated because Pac-Man has a couple of states and animations. There are 28 distinct modules that make up Pac-Man, one for each animation frame for each direction (except the solid circle) and 11 frames for the dying animation. The top-level module is implemented as nested state machines. The main state machine tracks whether Pac-Man is playing or dead. The inner state machine for the death sequence sends Pac-Man through eleven frames. The inner state machine for Pac-Man's typical "playing" mode controls how wide Pac-Man's mouth is open. Each state has one version of Pac-Man at its level of mouth-openness for each possible direction Pac-Man could be facing. The Pac-Man module includes three more inputs, `direction`, `start` and `dead`, so the gameplay logic can control its direction, when it starts playing the game and when it starts its death sequence.



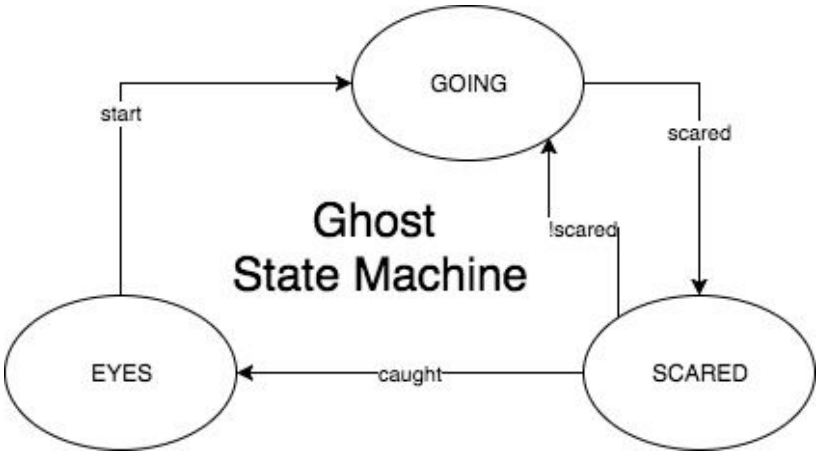
The ghosts work similarly to Pac-Man. Like Pac-Man, they require additional inputs to inform the graphics about the ghost's state in the game. Each ghost needs to know its `direction`, when it should `start`, when it is `scared`, and when it is `caught`. Direction works the same way as for Pac-Man (the ghosts' eyes always point in the direction the ghost is traveling). Ghosts switch to their scared graphics as long as scared is high, then to their caught mode (only showing the ghosts' eyes) when they are in scared mode and caught pulses. The bottoms of the ghosts are animated by toggling a single bit called `triangle_graphics` at approximately 12 Hz. The ghost's color is determined by a 12-bit `color` input. Ghosts are also capable of blinking when their period of being "scared" is almost done, but that feature was not used in the final product.



`triangle_graphics` high



`triangle_graphics` low



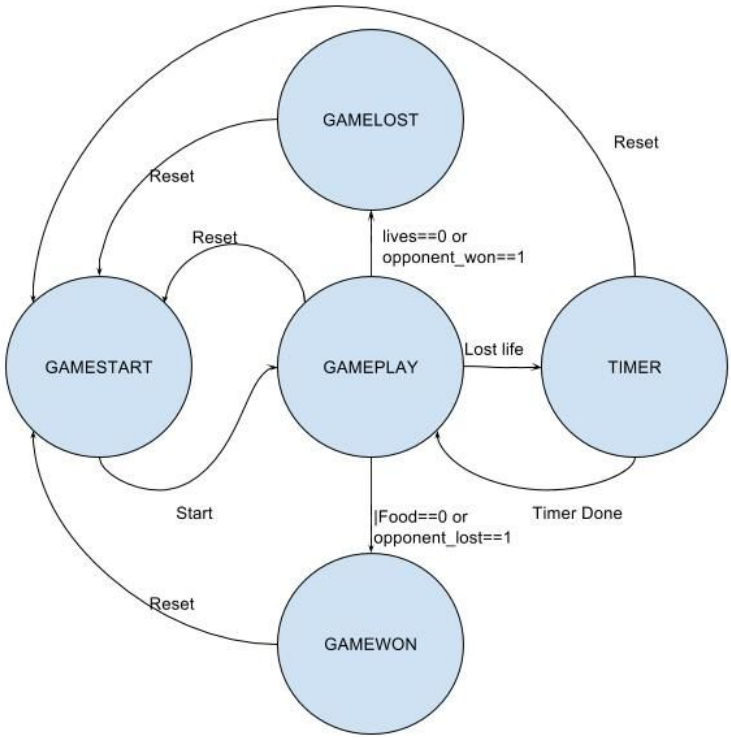
One challenge we encountered with these nested modules (both Pac-Man and the ghosts) is that clocking the transitions between states therefore shifting pixel outputs through multiple registers meant that the images were shifted one or two pixels to the right, depending on how many modules a pixel output had to go through before reaching the top level. To solve this

problem, the lower level sprites (for instance, the module of Pac-Man with its mouth completely open and facing right) calculate one or two pixels ahead of the current one.

Graphics that were created but not used include point values and many of the letters and numbers. We did not use the point values because we ended up not keeping track of points for the game. Luckily, because of the sprite maker, the time cost of making these extra graphics was very low.

Gameplay Finite State Machine - Rachael

The gameplay FSM is the fundamental setup that guides the other pieces of the project. The implementation can be found in game_fsm.v. The game has 5 states that can be found in the state transition diagram below:



This module controls the state of the game. GAMESTART gives the user control of when the action begins. GAMEPLAY mode is normal game; this mode is only left when a player loses a life or wins the game. TIMER mode is activated when a life is lost but the player isn't dead. (Players start with multiple lives.) This mode is exited on a timer and gives time for the dying animations and resets the characters to their original positions. The other two states are GAMEWON and GAMELOST. They are entered if all of the player's food is eaten, they have lost all of their lives, or they have outlasted their partner.

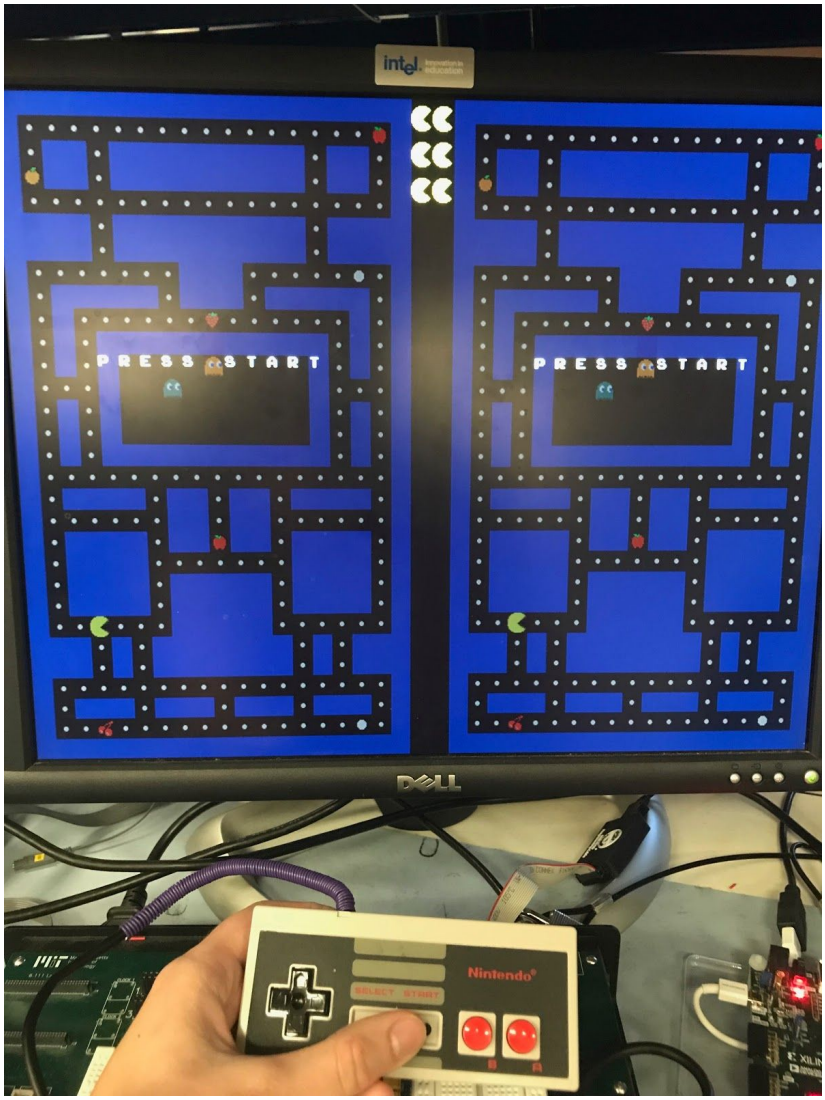


Image of game during GAMESTART

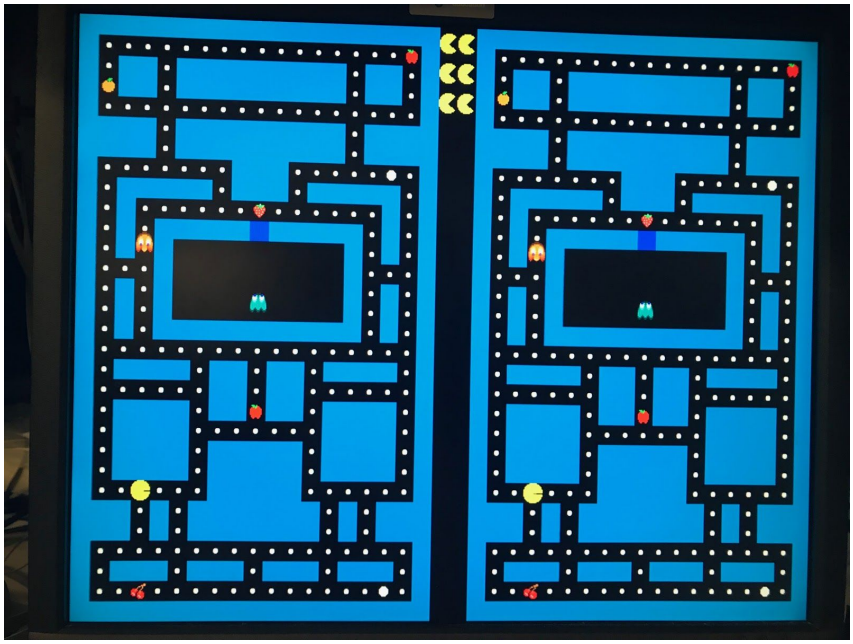


Image of game during game_play.

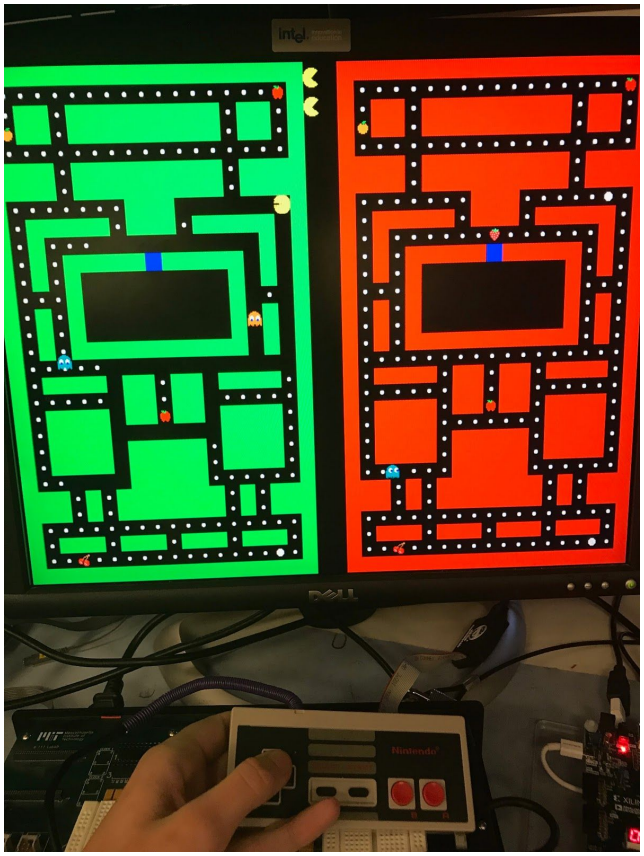


Image of end of game where player 1 (left) has won and is in GAMEWON and player 2 (right) has lost and is in GAMELOST.

There is another state managed by the gameplay FSM, the `special_state`. This state tracks which “special” is activated. When Pac-Man eats a special the effect of that special remains active until Pac-Man eats a new special or until five seconds have elapsed. When Pac-Man encounters food, the gameplay FSM actives the appropriate special state and the timer and removes that piece of food from the map. The gameplay FSM detects a food has been eaten if Pac-Man and food have non zero pixel outputs at the same time. Then, the FSM removes the food from the index of the collision and activates special as needed. These specials are expanded upon in the Power-ups section below.

The map is held in two 2D bit arrays. This was one of the most tedious and annoying things to set up and keep track of through the process. At first I made one map for the walls, one for the food, and one for the first powerup. This seemed easier to start, but quickly became large and difficult to track, display, and recreate. I transitioned to having the food map, which contains all information about edible objects, and the wall map, which just had details about the locations of the walls.

However, we still couldn’t pass the map information as inputs to modules. Therefore, if we wanted information about the current state of a cell, we had to get the information in the FSM and then pass it into the requesting module instead of the module getting to look up whatever it wanted. This situation made the code slightly messier and more cumbersome but no less accurate.

Finally, once the level builder was complete we needed to be able to pass maps as inputs and output. However, 2D bit arrays cannot be inputs and outputs in Verilog, and creating an input for each line of the array would have required 30 inputs per map on an input or output. This was unsustainable. We found that with consistent naming we could make a block of text that assigned all 30 of the rows to a large bus, pass that block in and out of modules, and then break it back out into a 2D bit array for indexing. This was effective.

The map is divided into cells of 16px x16px. This makes indexing into the map array easy, as the index of a location in the map is its location in pixels shifted right by 4 bits. For this implementation we have a 20 x 30 bit binary array for walls where a 1 represents a wall and a zero represents a non-wall. We also have an 80 x 30 bit array for food and specials where there is a nibble of information for each cell to indicate which food or special powerup is in each cell. The food is a nibble per cell so when the line was written in hex it is easy to determine what is in each cell. The food and wall were put in different arrays to address concerns that the gameplay FSM could accidentally delete or create walls if is was stored in the same data structure as the food. After implementation, we recommend combining the food power-ups and walls into one 2D bit array to save on wires, lookup tables and complexity.

Finally, the gameplay FSM controls the motions of the characters (Pac-Man and the ghosts). They are controlled by AI or user inputs. The gameplay FSM has an instantiation of movement.v

for each character that moves it in its desired direction once that direction is available. This was by far the hardest part to setup. At first I wanted to use the pixels as collision math. If the character and the wall were both not 0 at the same time, then the character should be moved back to the cell in the direction it came from. This is tricky, however, because you only know about a collision after it happens, and depending on when the button is released, the character may get stuck in a wall and jump randomly at the next button press. To solve this, the end module checks the walls map for directions a character can move in its next turn and only changes the character's direction if that direction is available to the character.

Powerups - Rachael

The powerups are managed by the gameplay FSM. The type of special in each cell is maintained in the food map. Since there are 4 bits per cell and empty space and normal food are available, there is the capacity for 14 different powerups in the game. For this implementation, we created 6 powerups, as we saw diminishing returns with creating more of them. For future work we encourage people to come up with more devious powerups than the ones we included.

The powerups are displayed in display_food.v. This works by creating one instantiation of each of the powerups' graphics. Each of the graphics has a location at the top left corner of the current cell of hcount and vcount, and only the graphic indicated by the food map is displayed.

The powerups we currently have mostly change speed or direction of Pac-Man or the ghosts. A list of the powerup effects can be found below:

1. Classic Pac-Man powerup: The ghosts become "scared" and Pac-Man can eat them.
2. Cherry: Your opponents controls flip. Left becomes right and up becomes down, and vice versa.
3. Key: Your ghosts become slow. They move at half speed and can't chase you as well.
4. Orange: Your opponent becomes slow. Your opponent moves at half speed and becomes much easier food for the ghosts.
5. Apple: You gain an extra life.
6. Strawberry: Your game pauses and you become your opponents ghosts. You can hunt down your opponent personally.

Enemy AI - Kim

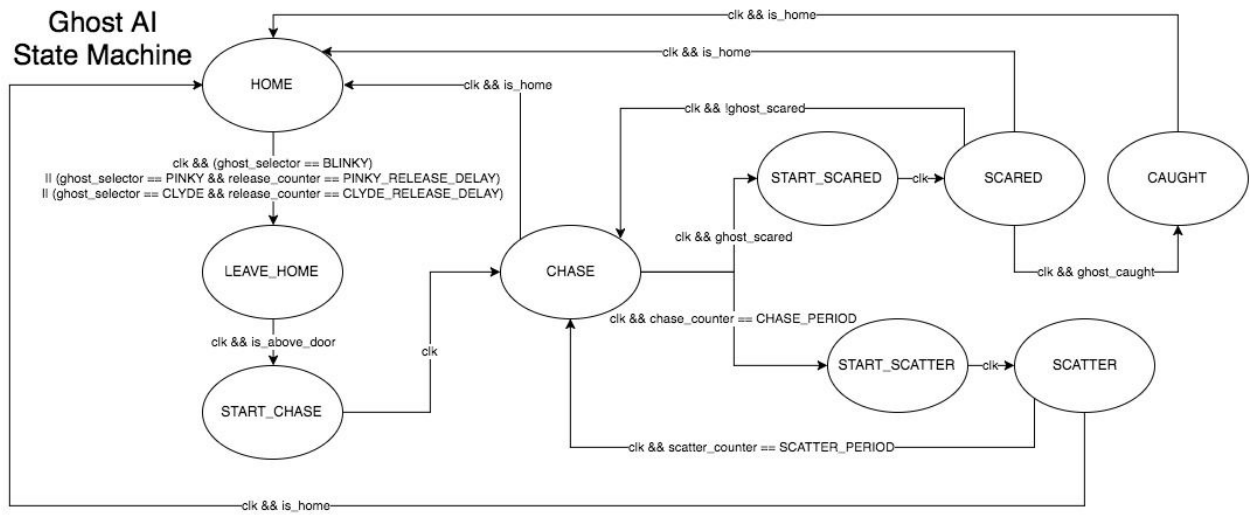
The ghosts' AI in this version of Pac-Man does its best to imitate the original game. Our main source of information about that AI was Chad Birch's very well-written article from 2010 about "Understanding Pac-Man Ghost Behavior."[2] The original creator of Pac-Man, Toru Iwatani, put a great deal of thought into how the ghosts move in order to make the game hard enough to be fun without being too hard to win. In particular, the ghosts have three different modes: Chase,

[2] http://gameinternals.com/post/2072558330/understanding-pac-man-ghost-behavior

Scatter, and Frightened. In Chase mode, the ghosts chase Pac-Man. In Scatter mode, they reverse direction and “chase” a corner of the board, giving the player a brief respite. In Frightened mode, the ghosts move around the game pseudorandomly.

The original four Pac-Man ghosts (Blinky, Pinky, Inky, and Clyde) all use different strategies to chase Pac-Man down. Blinky travels directly toward Pac-Man’s current location. Pinky targets a spot four “tiles” past Pac-Man’s current location in the direction Pac-Man is currently facing. Inky takes the vector from Blinky’s location to Pac-Man, doubles it, and targets the tile at Pac-Man’s current location plus that vector. If Clyde is further than eight tiles away from Pac-Man he chases Pac-Man directly; otherwise, if he is closer than eight tiles, he chases his corner. Due to time constraints we implemented the three simplest ghosts, Blinky, Pinky, and Clyde, because their movement depended only on their own current locations and directions and Pac-Man’s current location and direction.

At a high level, the ghosts’ AI is a finite state machine that changes how the ghosts move based on their state. In hindsight, this state machine assumes (sometimes incorrectly) that ghosts always reach home after being caught, so there should also be a route from `CAUGHT` to `CHASE` when `clk && !ghost_scared`. However, this diagram is the current implementation of the state machine.



For reference, `release_counter`, `chase_counter`, and `scatter_counter` all increment once per second. The `is_above_door` and `is_home` variables are updated every clock cycle and check whether the ghost is in the space above the door to the ghosts’ home box and whether the ghost is inside of the home box. The `ghost_scared` and `ghost_caught` variables are inputs to the AI. The following describes how each ghost behaves in each state.

- **HOME**: This is a ghost’s starting state for the game. It assumes that the ghost is in the home “box.” The ghosts’ releases are staggered based on the ghost. Blinky leaves

immediately, Pinky leaves after three seconds (PINKY_RELEASE_DELAY), and Clyde leaves after six seconds (CLYDE_RELEASE_DELAY). The ghosts bob up and down inside the box while waiting to be released.

- **LEAVE_HOME**: When a ghost is released, it must leave its home to go chase Pac-Man. In this mode, the ghosts move up and out of the home box.
- **START_CHASE**: This is simply a filler state to reset the counters and dividers before the ghost starts to chase Pac-Man.
- **CHASE**: This is the state the ghosts spend most of their time in. The AI updates the tile the ghost is targeting based on which ghost is being controlled. Ghosts' movement toward the tiles they are targeting is explained below in more detail. Ghosts stay in Chase mode for seven seconds.
- **START_SCATTER**: Ghosts must reverse their direction when they enter Scatter mode.
- **SCATTER**: In Scatter mode, the ghosts each target a different corner of the screen. They stay in Scatter mode for three seconds.
- **START_SCARED**: Ghosts must reverse their direction when enter Scared mode.
- **SCARED**: In Scared mode, the ghosts move pseudorandomly around the board. The pseudorandom movement is described in greater detail below.
- **CAUGHT**: If a ghost is Caught, it tries to go back to its home. If it becomes not scared before it reaches home, it goes back into Chase mode. Otherwise, it goes into the Home state when it reaches home.

A tricky problem we encountered when putting together this AI was that the ghosts calculated the next direction they should turn too fast in the Chase and Scatter modes. In particular, ghosts in Pac-Man are not permitted to reverse their direction directly in those modes (go from UP to DOWN or LEFT to RIGHT or vice versa). However, since 25 MHz is much faster than the ghosts' movement rate of one or two pixels per screen refresh, when the ghosts reached a juncture they could change directions twice before changing their location, resulting in an effect that looked like an illegal about-face. To solve this problem, we had the ghosts calculate their new direction at 400 kHz, barely slower than the screen refresh rate. If we redid this project I would have the AI update the next direction the ghosts should turn on the first time a ghost changes its position after changing directions. However, refreshing at 400 kHz works well enough for our purposes.

Targeting Algorithm

One of the trickiest parts of the AI was determining how to make the ghosts turn based on targeting a particular tile. This algorithm is used in the Chase, Scatter, and Caught states. It has three steps:

1. Sort the four directions (up, down, left, and right) based on which corresponding adjacent location is closest to the target tile.
2. Exclude the directions a ghost is not allowed to turn according to the rules of Pac-Man.
3. Choose the direction the ghost is allowed to turn will take it closest to its target tile.

The implementation of these three steps was not trivial, so we will now go into some detail on those implementations.

The AI first produces a list of the four directions (up, down, right, and left) sorted by the "proximity" of each adjacent location to the target. "Proximity" is simply defined as $|x_2 - x_1|^2 + |y_2 - y_1|^2$ for any two locations (x_1, y_1) and (x_2, y_2). We can use "proximities" instead of actual distances between locations because we only need to determine which of two potential locations is closest to a target. Specifically, if we know that

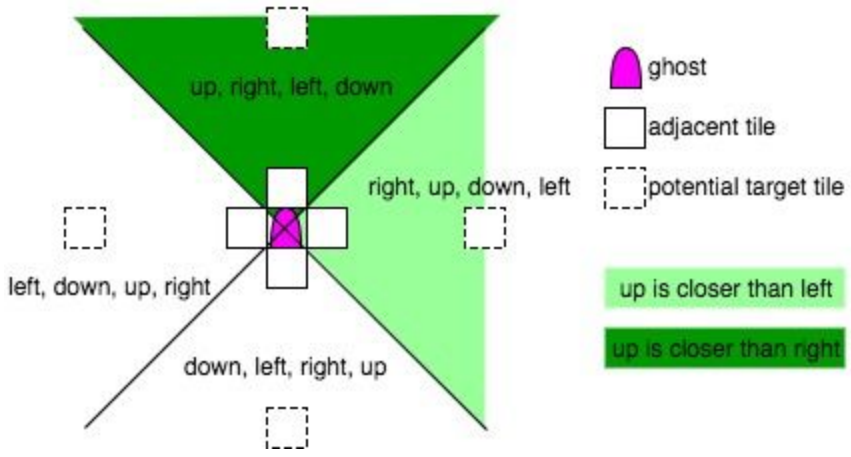
$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} < \sqrt{(x_3 - x_1)^2 + (y_3 - y_1)^2}$$

we can conclude that

$$(x_2 - x_1)^2 + (y_2 - y_1)^2 < (x_3 - x_1)^2 + (y_3 - y_1)^2$$

That is, (x_2, y_2) is closer to (x_1, y_1) than (x_3, y_3) is. Calculating a "proximity" takes three clock cycles, but this does not affect the graphics because the frame refresh rate is much slower than the 25 MHz clock. It's also several orders of magnitude faster than the 400 kHz at which the AI updates the next direction a ghost should travel.

Now that we can compare locations, we can generate the sorted order of the different directions' proximity to the target. We can do this in two comparisons that determine the "quadrant" the target location falls into relative to the ghost's current location and select the appropriate ordering based on that quadrant. The following diagram demonstrates what those orderings are in each quadrant.



Once the directions are sorted, we have to make sure we choose a direction that the

- Open directions (where there are no walls)

- Not directly opposite the ghost's current direction of travel
- Unless we're in the Caught state, not going back into the ghosts' home

To complete the algorithm, we simply choose the direction of travel that appears first in the sorted list and does not break any of the rules. This can be done relatively trivially with bitwise operators because of how we represent directions. Each "direction" is represented as a four-bit number with one bit high:

DOWN	0001
UP	0010
RIGHT	0100
LEFT	1000

The open directions are represented by four bits with the high bits corresponding to the open directions. For example, if DOWN is the only open direction, the open directions are 0001. If DOWN and LEFT are open, the open directions are 1001. The sorted directions are represented as 16 bits of the four directions with the closest-proximity direction in the most significant bit. For instance, if the correctly sorted order is DOWN RIGHT LEFT UP, the sorted directions are 0001_0100_1000_0010. Therefore, to determine whether the ghost can take the closest direction, for instance, we can simply check the following expression:

```
   (is_above_door && (sorted_directions[15:12] != DOWN)) // not above the door and going down
&& (sorted_directions[15:12] != opposite_direction)      // not going the opposite direction
&& |(sorted_directions[15:12] & open_directions)         // direction is available
```

The first direction to satisfy those requirements is the direction the ghost will go next.

The values required to perform these calculations, such as $x_2 - x_1$ and $(x_2 - x_1)^2$, are constantly recalculated in parallel, as are `is_above_door` and `is_home`.

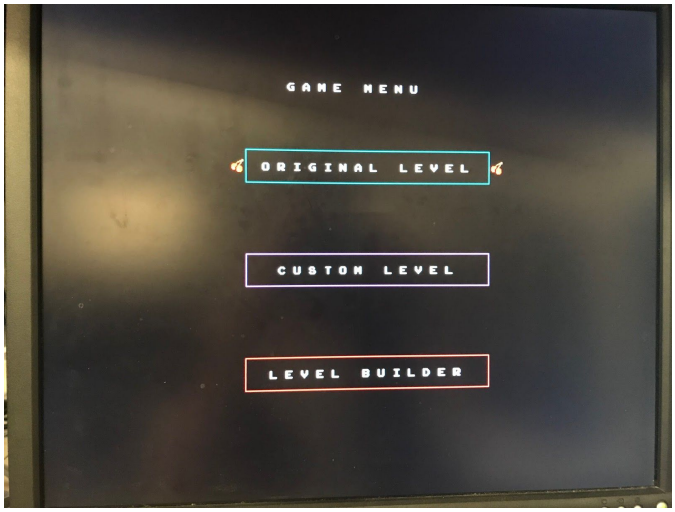
Pseudorandom Movement Algorithm

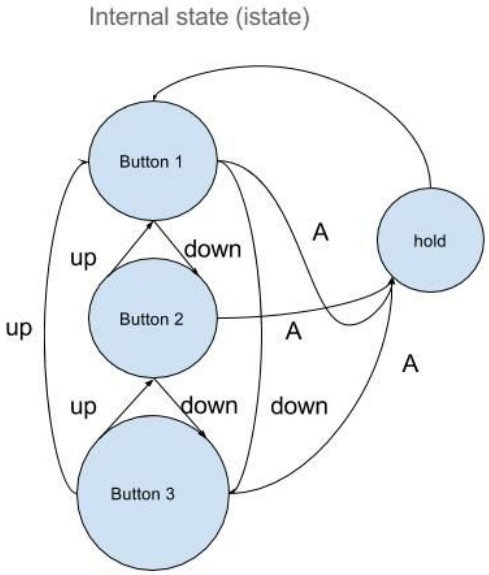
When the ghosts are in Scared mode, they have to move pseudorandomly around the board. I decided to use Pac-Man's current location as the seed for this randomness because the player is in control of that. Specifically, to produce a direction, we XOR the last two digits of Pac-Man's x coordinate with the last two digits of Pac-Man's y coordinate, the fastest changing digits of Pac-Man's location. Then, we use that result to select one of four possible orderings of the four directions. There are 24 possible orderings total, but it only takes four orderings for each direction to appear in each position in the order. The ghost's location on the board and the directions available due to that position add another element of randomness.

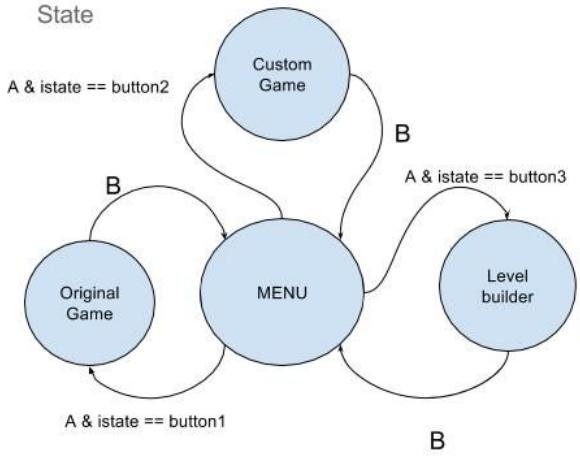
The main problem we ran into when generating randomness this way is that Pac-Man occasionally stops moving when he runs into a wall, which causes the ghosts to use the same ordering of directions while Pac-Man is in that location. To combat this, I introduced a two-bit selector that increments every clock cycle. This introduces enough variety that the ghosts move unpredictably while scared.

Game Menu - Rachael

The games menu manages two small FSMs. Its implementation can be found in game_menu.v. One three-state internal FSM manages the location of the cursor on the screen (the cherries in the image below). The other FSM has four states that manage what mode the user is in: gameplay default game, level builder, gameplay custom level, or game menu. Both of the finite state diagrams can be found below.







The internal state is managed by the user controls. The state changes as the user scrolls up and down through the menu with wraparound. When the user has chosen a state with the A button, the internal state remains locked in a hold state until the state is returned to the MENU. At that point, the internal state is set to BUTTON1 and scrolling can continue.

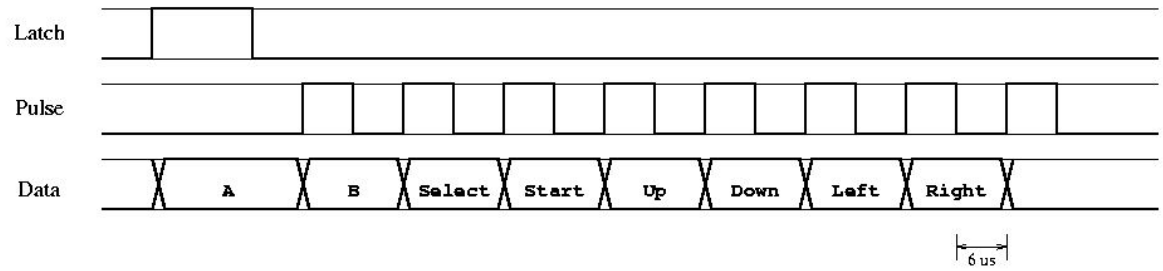
The state is managed with user input as well. When the state is at the MENU, it exits that state when A is pressed. Its location depends on the internal state. It remains in that state until B is pressed and then returns to the MENU. While state is in MENU the reset to a game is held high so a new game is started whenever a game state is chosen from the menu.

Beyond the logic, the game menu uses a lot of graphics modules to create letters and buttons.

As a final note, we included both a system and a game reset. The system reset resets all the modules and returns the user to the main menu. Game reset is activated by system reset but is also held high while the state is MENU or LEVEL BUILDER to ensure a new game is started when a game is selected and no nonsense inputs mess with the game. The user also has access to game restart so they can restart a game without switching state or losing a custom-built level.

Controllers - Rachael

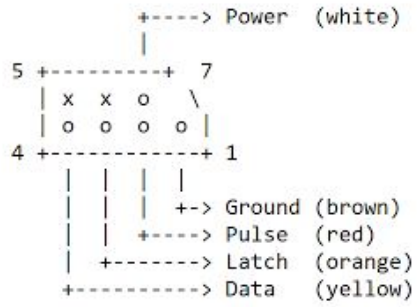
We implemented the use of two NES controllers with the nes_controler module. The controllers take two inputs of latch and pulse and output data on a shift register. Here is the timing diagram for the NES controllers input and output pins[3]:



Where the controller looks like:

[3] https://tresi.github.io/nes/

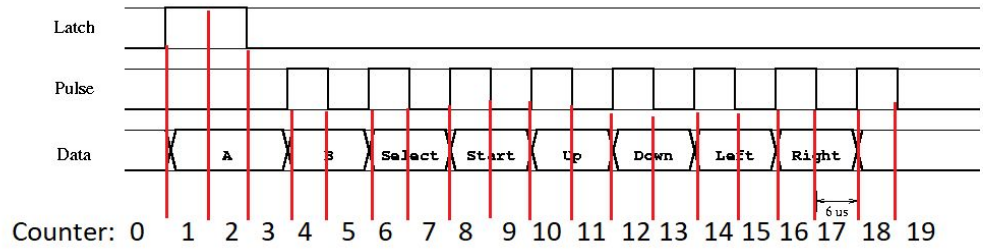
The pins of the controller are oriented as shown below:



NES Controller Pins

Where the Power pin is connected to 3.3V on the FPGA, Ground is connected to ground on the FPGA, and Data, Latch and Pulse are connected to JA. For this to work, JA must be specified as an inout port so that pulse and latch can be outputs of the FPGA and data can be an inout to the FPGA.

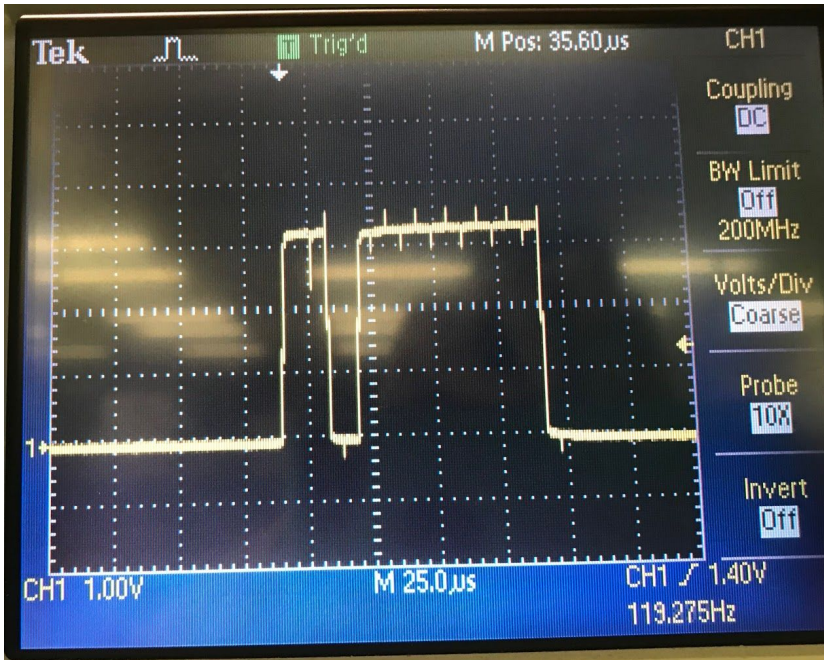
We created a finite state machine which has a counter as its state.The module waits for a start pulse to begin polling for data. In the game we poll for data every screen refresh as we won't use the data more often than that. The counter increments every 6us that the module is active. This means the counter has values as shown below in the timing diagram, where each red line is 6us apart:



Counter becomes 0 at the start pulse and the counter is not incremented when counter >= 19. The values of latch and pulse are based of the counter where latch is high when counter is 1 or 2 and pulse is high when counter is even and between 3 and 19. The data collection is also based off the counter. Data is guaranteed to be stable on the falling edge of pulse. Therefore we sample data when counter is even and we are about to increment it to an odd number and counter is between 2 and 17. We intake the data in a shift register, and after the data pooling is done we put the data in a output register. This means the data on the output is always clean and in the correct order.

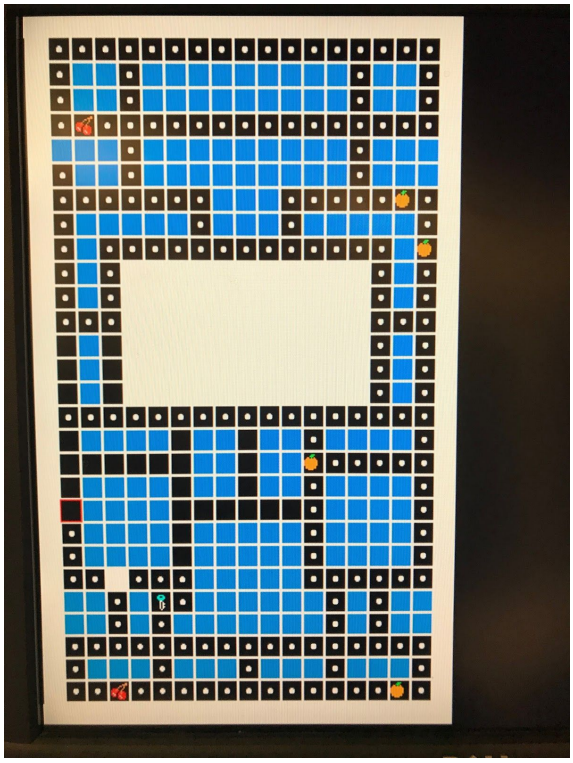
We increment the counter every 6us. Therefore a data_clk was created such that it has a one cycle pulse every 6us. Counter is incremented when counter < 19 and data_clk is high.

The last note about this controller is that data is low until a latch pulse is asserted, then is active low when a button is pressed. See image below of data line when B is pressed.

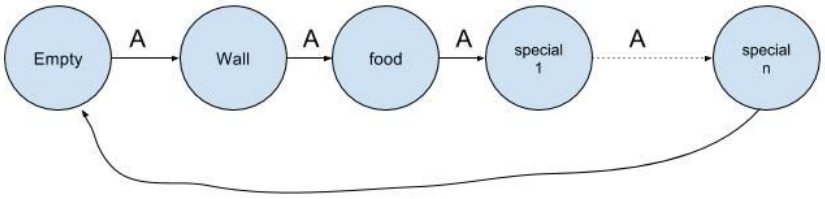


Level Builder - Rachael

The level builder uses the map structure to its advantage by allowing users to place walls and food almost anywhere they would like. First the level builder makes a grid on top of the default map (using grid.v) so the user knows where all the spaces for an object are. The level builder uses a simple FSM to keep track of the user's current location and highlights that location in red - see image below where the selected cell is (1, 20).



The level builder starts with the default map displayed. The user uses the cross on the controller to navigate to different locations and uses the A button to change what is present in a location. Each time the A button is pressed, the state of that location changes according to the state transition diagram below:



Where there are n specials and each time A is pressed the cell selected increments to the next special until it reaches special n and then it follows the diagram back to being an empty cell.

There are a few cells that do not follow this progression. The walls on the edge of the board and the location of the ghost's home and Pac-Man's starting location are shown in white, and the user is unable to change those objects. This is because Pac-Man is unable to travel outside the bounds of the screen and because the ghost and Pac-Man's starting location are fixed points, so the user would disrupt game play by trying to move them. These are all listed as location blocked cells, and the level builder will not change their state.

This level is output to the game FSM in the form of two buses, food and walls. Food is a 2400 bit bus and walls is a 600 bit bus. They are constantly up to date with the output on the screen.

When the user selects to play the custom level they built, the level builder's output replaces the default map as the input for walls and food. The game runs identically otherwise.

Final Notes

Our project encountered a few problems with integration, mainly in somehow using too many resources on the FPGA. At first we used too many lookup tables; then, after some editing, we used too many wires between modules. In the end we almost fixed this by passing the default map into level builder as a parameter. In theory this fixed the too-many-wires problem, but much to our chagrin Vivado claimed our movement.v was using 300 lookup tables per instantiation, which easily took up all of our available lookup tables. To free up these lookup tables, we could only have 2 of our 3 ghosts on the screen. I don't think there are 300 wires in one instantiation of movement.v, so I can't believe that that was the cause.

Even more perplexingly, if we set the ghosts' pixel to

```
ghosts_pixel == blueghost_pixel | greenghost_pixel | pinkghost_pixel;
```

the code would fail at implementation because we used over 100% of the lookup tables. However, if we changed only that one line to

```
ghosts_pixel == blueghost_pixel | greenghost_pixel;
```

the module compiled and ran smoothly and we only used 89% of the lookup tables...but the pink ghost was invisible. We decided this invisible ghost was a feature of the game and that we have found some strange esoteric vivado bug.

Many thanks to the 6.111 staff, especially Gim, Joe, Weston, and Maddie for helping us out with our project and teaching us so much over the course of the semester. Overall, we are delighted with the way our final project turned out. We've loved showing it off to our friends, and we were pleasantly surprised at how much people wanted to play the game again and again after they tried it once. Most importantly, this project satisfied our innate desire to vanquish our opponents through tricks and skullduggery - the best end result we could have ever hoped for.