Cassie Huang and Michael D'Ambrosio
6.111 – Introductory Digital Systems Laboratory
December 14, 2005

For our final project, we developed Pac-Man on the FPGA. The game contains modules to handle both the display of the Pac-Man world and logic to control all of the characters and define the rules of the game. Cassie predominantly worked on the display portion of the game and Mike mostly worked on the game logic portion of the game.

Essentially, all display modules revolve around drawing objects from memory. The map, the dots, the ghosts and Pac-Man all had separate ROMs which dictated how to draw each corresponding sprite on the VGA display. The lone exception was the dots, which involved a RAM, as we wished to eat them at some point and make them disappear from the screen.

We made it such that Pac-Man runs around a maze while avoiding the ghosts, all of which were controlled by separate AI modules. Pac-Man would die if he touched a ghost. However, he could eat an Energizer and temporarily eat the ghosts. A game over occurred if the player died three times but if the player ate all of the dots on the screen, they would win and be treated to a humorous animation.

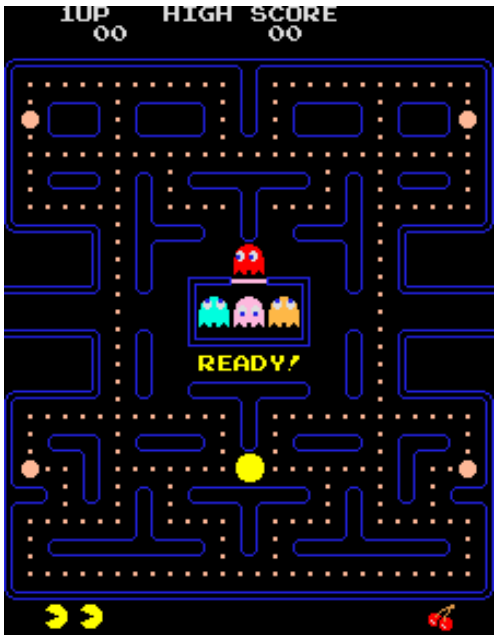
As a result of creating this project, we were able to faithfully recreate a classic arcade game. We also learned how to work as a team in order to build a complex digital system from the ground up.

Table of Contents

List of Tables and Figures

1. Introduction

Pac-Man was created in 1980 by Namco. The idea came from Namco employee Ioru Iwatani[1]. One day, he was eating pizza and noticed that a pizza pie with a slice missing looked like it could be a cartoon character. Namco wanted to create a game that would appeal to many different age groups so they decided to base a game on maneuvering a maze and eating. The resulting game was originally called PUCK MAN, which stems from the Japanese word *pakupaku* which literally means, "to flap one's mouth open and closed." When the game was first released in Japan, it received a decent response, but nothing terribly exciting. As a result, Midway picked up the license and brought the game over to America. The title was changed to Pac-Man to prevent the most obvious vandalisms on the old name and the cabinet artwork was redesigned. The game was a huge success and many spin-offs and other versions resulted.



Essentially, Pac-Man is a game of hide and seek. The player controls the yellow blob that is Pac-Man and maneuvers him throughout the maze. The object of the game is to eat all of the dots on the screen while avoiding the four ghosts that patrol the maze. Since Pac-Man is outnumbered four to one, there are four larger dots, called Energizers, which allow Pac-Man to eat the ghosts for a brief period of time. During this time, the ghosts turn deep blue and run in the opposite direction. A player earns more points for eating a ghost and the points are cumulative, meaning that it is beneficial if a player eats all four ghosts before they turn back to their normal colors.

[1] All background history information on Pac-Man was taken from Wikipedia (http://en.wikipedia.org/wiki/Pac-Man)

Furthermore, there are fruits which appear every level. A player can eat the fruit to gain bonus points. Also, a player can gain lives by attaining a certain number of points.

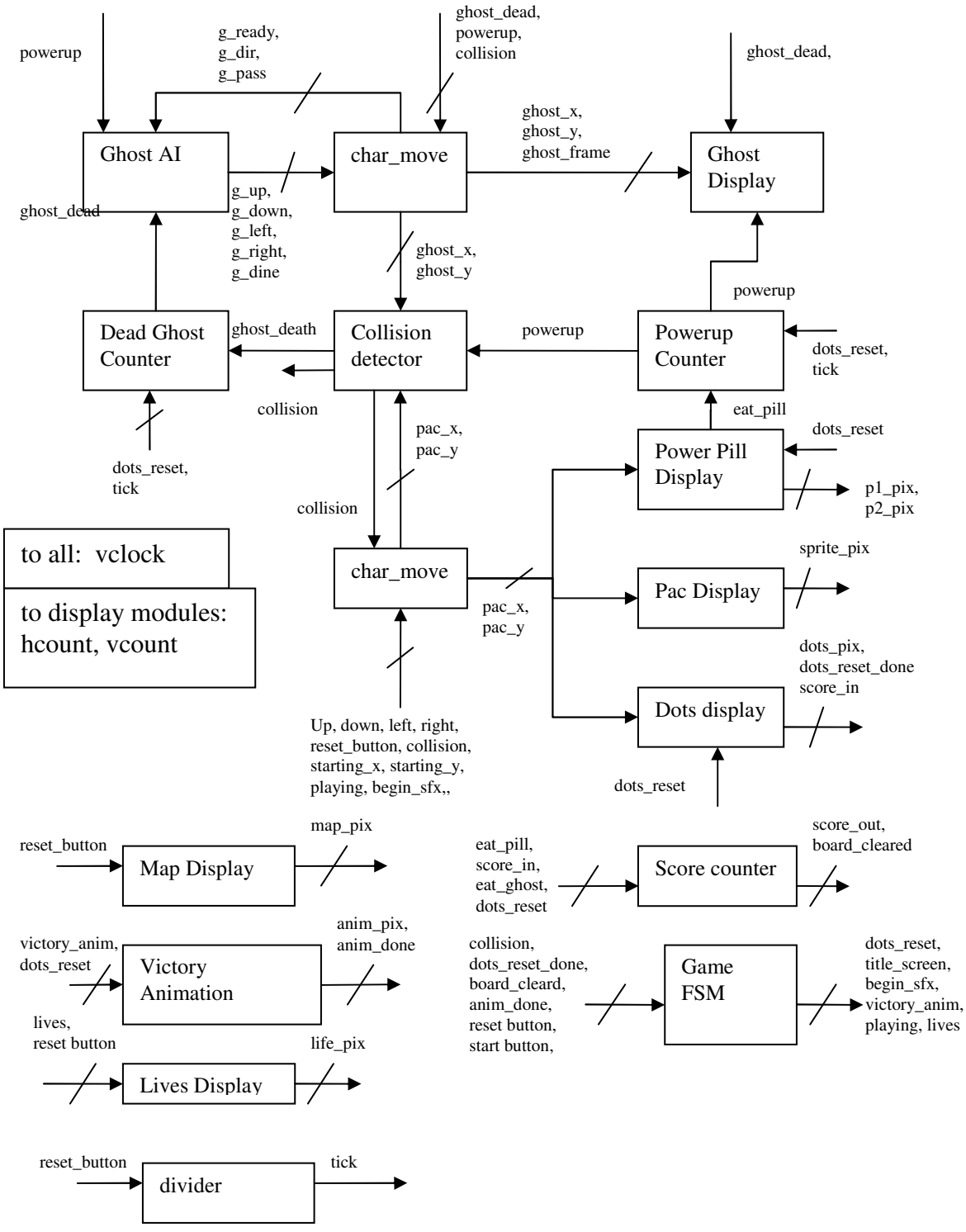


Figure 1b – Block Diagram: This figure represents the block diagram for the Pac-Man game in its most complete form.

For our final project, we sought to re-create the Pac-Man experience via the FPGA. Refer to figure 1b above for a picture of our final block diagram. We wanted to faithfully reproduce the features that the original game had. The user is presented with a VGA display and a keyboard. Upon startup, the game goes to its title screen, which proudly displays our names, the title of our project and the controls for the game. The user interfaces with the game via the keyboard. The enter key starts the game. The space bar is used to pause the game during play. The WADS configuration, as it is generally known as, is employed to control Pac-Man. The W key moves Pac-Man up, the A key moves him left, the D key moves him right and the S key moves him down. From the title screen, the player can press the enter key to begin the game. The block diagram of the project is shown above in Figure 1.

After pressing Enter, the game will draw the side display (consisting of a score counter and a life counter), draw the map, fill it with dots and place all the characters in their respective starting positions. For the four ghosts, they are placed within the "ghost pen," which is a 4x1 area in the middle of the screen and Pac-Man is placed near the bottom of the screen, about halfway across from either side. The game would then play a sound corresponding to the level starting. After this sound plays, the user could then move Pac-Man and eat the dots. The ghosts would move from their pen and behave as they have been told to. Each ghost has a unique strategy and those are described in greater detail later on. Located in the top right and the bottom left hand corner of the maze is a blue dot. These blue dots are Energizers. When the player eats them, the ghosts turn dark blue, they slow down, and they run away from Pac-Man, albeit slowly. If a ghost is ever eaten, they will disappear for some length of time and then reappear, emerging from the ghost pen and returning to normal.

For every dot that the player eats, they gain one point. For every Energizer that the player eats, they gain ten points. For every ghost that the player eats, they gain one hundred points. This value is maintained in a register and displayed on the VGA next to the maze for the player to see.

If a player ever touches a ghost and isn't in power pill mode, the mode where the player can eat ghosts, then the player dies. Ideally, the game transitions out of the playing state so that no character can move and a short sound is played corresponding to the player dying. After the music is done, the game will decrement the player's lives by one. If the lives are greater than zero, the game will then reset all of the characters to their starting positions and start the level again. In this case, the dots will not be reset nor will the score be reset. If the player's lives have dropped to zero, then the game will return to the title screen and all game states will be reset: lives, score, etc.

Furthermore, if a player manages to eat all of the dots on the screen, then a brief sound will play corresponding to victory. After this sound, the player will be treated to a short, humorous animation involving Pac-Man and one of the ghosts. After this animation, the game will reset the board, reset all characters' starting positions and the level will start again. The player's score and lives will be retained.

Lastly, there is a global reset button, the enter button on the labkit, which allows for the entire game to be reset at any time. Whenever the global reset is asserted, the game returns to the title screen and all states are reset.

2. Cassie's Modules and Implementation

I was responsible for designing the map and the sprites, and displaying everything on the screen. I had to design the map and dots displays twice, because I attempted to brute force the display with combinational logic and real time display instead of reading information from ROM and RAM. I was also responsible for Pac Man eating the dots as he moves around the screen, and the display and effects of the power pills. The last features I was responsible for are the score display and the ending animation.

2.1 Video Displays

The video display is a collection of modules that display the map, the dots, the ghosts, and Pac Man as well as the current score and the number of lives that Pac Man has. All display modules use XVGA. The video is displayed in real time though images are read from ROM. Each element is drawn independently and their outputs are connected using an or gate and sent to the screen. Each display modules takes as inputs the 65 MHz pixel clock, and hcount and vcount signals from the XVGA modules, and output a pixel value. Figure 2.1 shows a block diagram of the video display.

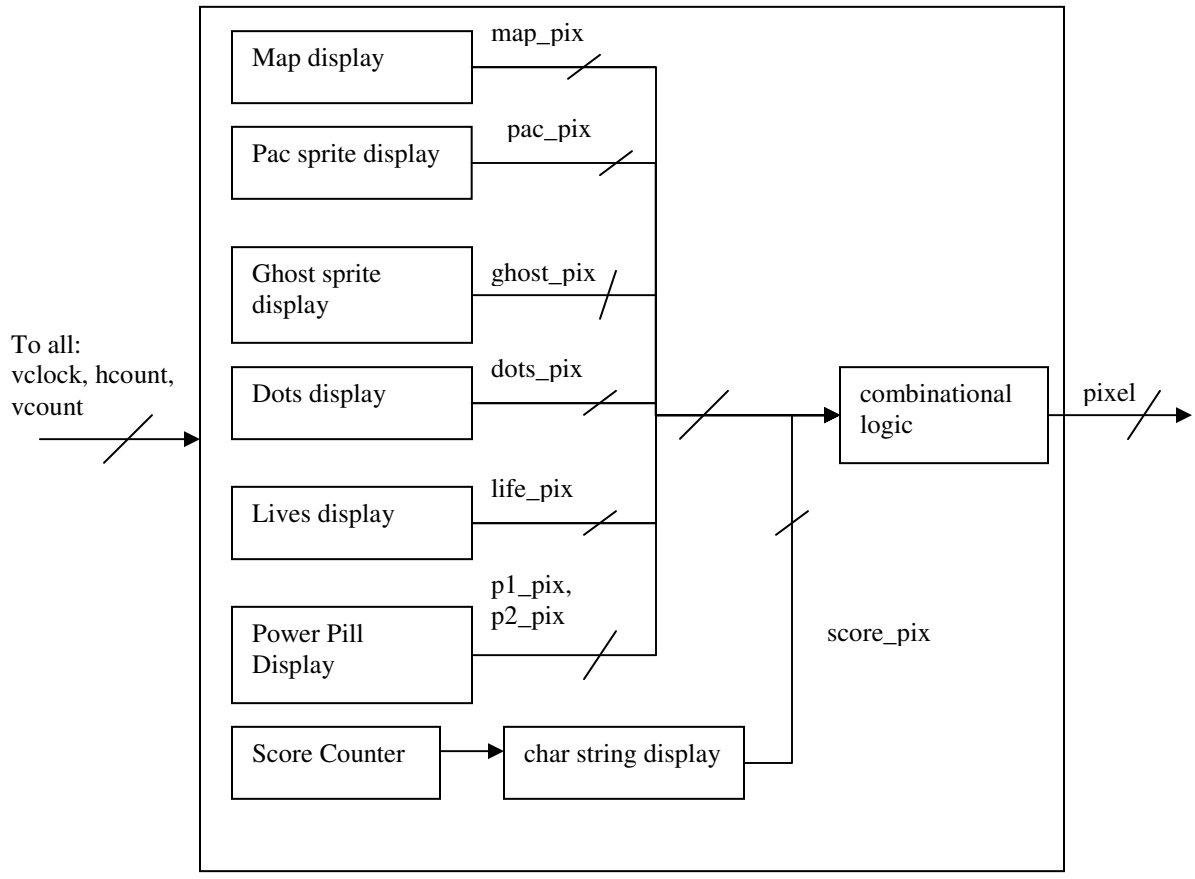


Figure 2.1 – Block Diagram of the Video Display
Each display modules takes in as inputs the pixel clock, hcount, and vcount, and outputs pixel data.

2.1.1 Map Display

The map is an arrangement of 32x32 pixel blocks, called tiles, of walls and blank space. The tile pattern of the map is stored in a 16x24 ROM, and read when the map is displayed. Each bit stored in memory represents a block. A bit that is 0 is a space tile while a bit that is 1 is a wall tile. Each bit is read 32 times horizontally and vertically to draw the tile.

Each row read from the ROM is a row of the map.The address given to the ROM resets to 0 when the entire map has been displayed, and hcount is equal to 1343 and vcount is equal to 805. The address increases once when vcount is equal to a multiple of 32. As each row is displayed, the current bit displayed is selected by the current value of hcount. The last five bits of hcount are ignored so the bit only changes when hcount is a multiple of 32. To center the map, which is 16 tiles wide, the map is only displayed when the higher 6 bits of hcount are between 8 and 24. The bit selected is then determined by the higher 6 bits of hcount subtracted by 8 to ensure the map is properly centered.

2.1.2 Dots Display

The dots display module handles displaying dots on the map, Pac Man eating dots as he wanders around, and refreshing the dots after a reset or after the board has been cleared. In addition to the inputs that all display modules take, the dots display takes as inputs a reset signal, the current x and y tile coordinates of Pac Man, and vsync from the XVGA module. It outputs score, a signal that is high for one clock cycle after a dot is eaten, and dots_reset_done, a signal that tell the game state machine that the dots have finished refreshing.

The dots display consists of a 16x24 RAM that contains the current dot locations to be displayed and a 16x24 ROM that contains the permanent locations of all the dots that will be loaded into the RAM on a reset. The RAM has three possible addresses depending on which function the dots display is asked to perform. A tristate buffer is connected to each possible address and the address port of the RAM to prevent more than one value driving the address port at a given time.

The first address is the display address, which selects the row in the RAM to be displayed on screen. This address is similar to the address of the map ROM. It resets to 0 when hcount is 1343 and vcount is 850, and increases by 1 when vcount is a multiple of 32. This address is only active when vsync is high, and reset is low.

The second address selects the row of tiles that Pac Man currently occupies, and is used to select the bit in the RAM to overwrite if Pac Man has eaten a dot. Since this address may need to be rewritten, the address is only active when vsync is low, and reset is low to prevent the RAM from being rewritten as it is being displayed.

The last possible address is the refresh address, which is only active when reset is high. This address selects the row in the RAM to be rewritten with data from the ROM.

Dots are displayed in a similar manner to how the map is displayed. Each bit that is 1 in the RAM represents a dot to be displayed on the screen. The display address selects the row to be displayed, and the upper 6 bits of hcount select the bit to be displayed. The dots display is centered like the map display is. Each dot is centered in the tile by only displaying the dot when the lower 5 bits of hcount and vcount are between 14 and 18.

On reset, data from the ROM is loaded into the RAM. A counter called the reset counter is activated. The reset counter determines the addresses for the RAM and for the ROM. The address of the ROM is one higher than that of the RAM because there is a lag of one clock period between when data is read from the ROM and when data is written to the RAM. Having the same address for the RAM and ROM results in the dots display being shifted by a row. Write enable to the RAM is activated one cycle after the data has been read from ROM. This is to prevent the RAM from writing invalid data. The signal dots_reset_high is activated when the reset_counter is 31.

During each cycle when vsync is negative, the dots display checks for eaten dots. The eat address is only active when vsync is negative, and selects the current row that Pac Man is occupying. The x tile coordinate of Pac Man is loaded into a register, and used to select the bit in the RAM that will be overwritten. A 2 bit counter is activated to separate process that would interfere with each other. When the counter is 0, the row just read from RAM is loaded into the write port of the RAM, dots_write. On the next count, the x tile coordinate is used to select the bit of dot_write. If the selected bit is a 1, Pac Man is in the same tile as a dot, and that dot is rewritten to a 0. The score is set to high. If the selected bit is a 0, nothing happens. When the counter is equal to 2, write enable for the RAM is set to high, and the updated data is written to RAM. Score is then set to 0 to ensure that it is high only for one cycle. After that, the write enable is set 0. When vsync is high, both the counter and write enable are set to 0 to prevent the RAM from being rewritten as its contents are being displayed.

2.1.3 Pac Man Sprite Display

The Pac Man sprite display takes as additional inputs the coordinates for the upper left pixel of sprite, the direction the sprite is facing, the current animation frame, and the negative edge of vsync. The sprite for Pac Man is stored in a set of 32x32 ROMs. Pac Man is only displayed if the pixel is within 32 pixels vertically and horizontally of the x and y coordinates given to the module. The x and y offset value represent how far from the upper left pixel the current pixel is.

Since Pac Man is nothing more than a yellow circle with a slice taken out, it is fairly symmetrical. The sprite for Pac Man can change which way it is facing given a direction by changing how the ROM is read. If the sprite is facing up or down, the row read from ROM is determined by the x offset with a shift of 3 added. A new row is read each time the x offset changes. If the sprite is facing left or right, the row read from ROM is determined by the y offset. The address increases by 1 each time the y offset is less than 32, and the x offset is 32. In all cases, the address is set to 0 when the x and y offsets are greater than 32. The pixel to be displayed is determined by the y offset if Pac Man is facing up or down, and the x offset if he is

facing left or right. The new direction for Pac Man is stored in a register at every negative edge of vsync so the sprite does not respond to direction changes in the middle of display.

There are 8 separate frames of animation for Pac Man. The animation cycles from the mouth being wide open, to the mouth closing, to the mouth opening again. Five ROMs store the images for the animation. All ROMs use the same address. The ROM to be read from is selected by the animation frame input.

2.1.4 Ghost Sprite Displays

Four instances of the ghost display module are created, one for each ghost. The ghost display takes as additional input the coordinates for the upper left pixel of the ghost to be displayed, and the animation frame. This module behaves very similarly to the Pac Man display module. The ghost sprite is stored in a 32x32 ROM. The ghost is only displayed when the current pixel is within 32 pixels both horizontally and vertically of the coordinates of the upper left corner of the sprite. The x and y offsets measure how far from the upper left pixel the current pixel is. The row of the ROM to be displayed is chosen by the y offset, and the bit of the row to be displayed is chosen by the x offset. Ghost colors are parameterized with the default color as red.

The ghost animation has 4 frames, each of which is stored in a separate ROM. The ghost will rotate its eyes and wiggle its feet when it is animated. The animation frame determines which ROM is read from. Ghosts will respond to certain signals if the Pac Man has eaten a power pill. Those effects will be discussed in the power pill section.

2.1.5 Lives Display

The lives display is a simple module that takes the number of lives Pac Man has from the game state machine, converts the number to an ASCII character using a lookup table, and displays that character using the module char_string_display. The modules locks in the number of lives Pac Man has at every negative edge of vsync so the characters to be displayed don't change while they are being displayed.

2.2 Power Pills

Pac Man is not left completely defenseless against the seeking ghosts. Two power pills, located at the upper right and lower left corners of the map, allow him to be invincible for a short period of time after eating them. The effects of the pills are as follows. If Pac Man eats a power pill, all the ghosts turn blue, and start to move away from him at a fraction of their normal speed. If Pac Man encounters a ghost after he has eaten a power pill, he can eat the ghost. Invincibility for Pac Man only lasts for 16 seconds. Ghosts that Pac Man eats will disappear from the map for 16 seconds, and then return. Power pills are implemented with three modules, and the effects of the power pills require changes to the collision detector and modules that control displaying ghosts, ghost movement, and ghost AI. Figure 2.2 shows the signals involved in implementing power pills.

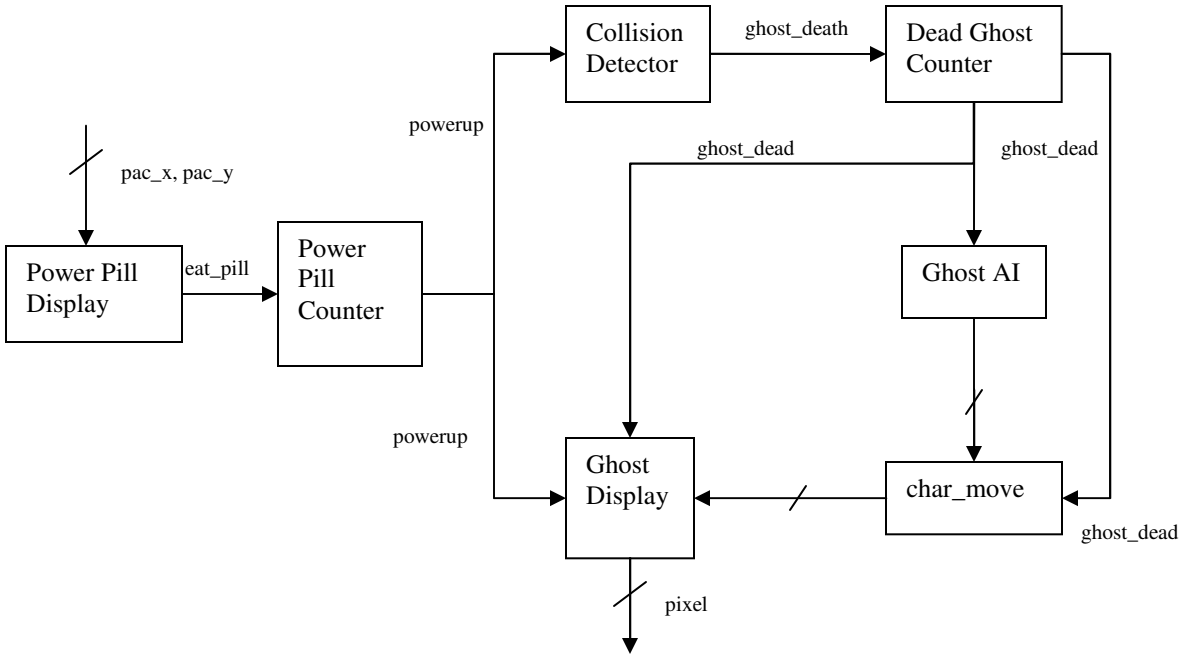


Figure 2.2 – Block Diagram of the Power Pills Modules
This shows the signals generated by the modules involved in implementing effects of power pills.

2.2.1 Power Pill Display

The power pill display module displays a power pill in the upper right and lower left corners of the map. This module takes as inputs the 65 MHz pixel clock, hcount and vcount from the XVGA module, and the current x and y tile coordinates of Pac Man. It outputs the pixels of the pills and the eat_pill signal, which goes high when a power pill has been eaten. The pill display refreshes on the signal dots_reset from the game controller FSM.

The pills are a pair of enabled blobs. Blobs display a block of color given the coordinates of the upper left pixel, and the size of the block. An enabled blob only displays a blob if the enable signal is high. If the tile coordinates of Pac Man match the tile coordinates of a power pill, then that pill becomes disabled, and not displayed on the screen. The pills are only regenerated during when the dots are reset. The eat_pill signal goes high when a pill is disabled, and lasts for longer than one clock cycle.

2.2.2 Power up Counter and Dead Ghost Counter

The power up counter counts the number of seconds Pac Man has been in power up mode. It turns power up mode on if it receives a high eat_pill signal, and turns power up mode off if 16 seconds have passed. There is an instance of the dead ghost counter for each ghost. The counter counts the number of seconds a ghost has been dead. The counter activates when it receives a

signal saying a ghost has died. The output signal ghost_dead is high as long as the counter is active. The counter turns off after 16 seconds.

Both the power up counter and the dead ghost counter receive a signal called tick from the divider module. The divider module generates a signal that is high for one cycle every second. The power up counter takes as other inputs the eat_pill signal from the power pill display, the 65 MHz clock, a reset signal, and outputs the signal powerup, which is high for 16 seconds after Pac Man has eaten a powerpill. The power up counter detects the rising edge of eat_pill because eat_pill is high for more than one cycle. At that rising edge, the counter is activated, and powerup is set to high.

The dead ghost counter takes as other inputs the ghost_death signal from the collision detector, the clock, and a reset signal. It outputs the signal ghost_dead, which remains high for 16 seconds. The dead ghost counter detects the rising edge of ghost_death because ghost_death is high for more than one cycle. At that rising edge, the counter is activated, and ghost_dead is set to high. Both the power up counter and the dead ghost counter set their timers and outputs to 0 when they receive the dots_reset signal from the game controller FSM.

2.2.3 Changes in Other Modules to Handle Ghost Death

When a ghost dies, it is moved back to the ghost pen, and not displayed or handed new moves until its dead ghost counter expires. Modules that control ghost movement and display required additional logic and inputs to handle ghosts being eaten by Pac Man and disappearing off the screen after they die. When a ghost dies, it is moved back to the ghost pen, and not displayed or handed new moves until its dead ghost counter expires.

The collisions module required the additional input of powerup, and the additional outputs of ghost_death for Inky, Pinky, Blinky, and Clyde. If powerup was high, then a collision would not be reported to the game state module if Pac Man collided with a ghost. Instead, the collision detector would check to see which ghost collided with Pac Man, and send a ghost_death signal to the corresponding dead ghost counter.

The movement module and the ghost AIs added the inputs powerup and ghost_dead. The movement module sets the speed the ghosts to 1 if the powerup is high. When a ghost is dead, the movement module behaves as if it was in the reset state, and does not move the ghost. The ghost AIs will tell the ghosts to move away from Pac Man if powerup is high. When the ghost_dead signal is high for a ghost, that particular AI will stop generating moves for that ghost, and behaves as if it was in the reset state.

The ghost display modules also take in powerup and ghost_dead as additional inputs. When in powerup mode, all the ghosts change their color to blue. When ghost_dead is high for a particular ghost, the ghost display module for that ghost no longer draws the ghost on screen.

2.3 Scoring

The player earns one point for each dot Pac Man eats, ten points for each power pill, and one hundred points for each ghost. The maximum score the player can earn for a game is 999 points. The score counter modules keeps track of the current score, and converts the score to a series of ASCII characters to be displayed. The module takes the signals score from the dots display module, eat_pill from the pill display module and eat_ghost, a signal generated by connecting all 4 ghost_death signals from the collision module with an or gate, to generate the score. It takes as other inputs a reset signal, the clock signal, and the negative edge of vsync. This module outputs the signal board_cleared, which tells the game controller FSM that Pac Man has eaten all the dots on the screen, and score_char, the numerical score converted to ASCII characters. The score cleared each time it received the dots_reset signal from the game controller FSM.

Edge detectors are used on the input signals eat_pill and eat_ghost since those signals are longer than one clock cycle, and the score should only respond once to a high signal. The score module has three four bit registers that represent the hundreds, tens and ones digits of the score, hscore, tscore, and oscore. An 8 bit register that counts the total number of dots eaten. The dots eaten register is incremented by 1 every time the module receives the score signal from the dots display. The signal board_cleared is set to 1 when the dots eaten register is equal to 179, the total number of dots on the board. The ones digit, oscore, increases by 1 each time the module receives the signal score. When oscore is equal to 9, it is set to 0 on the next clock cycle, and tscore is incremented by 1. When oscore is equal to 9 and tscore is equal to 9, both are set to 0 on the next cycle, and hscore is incremented by 1. The tens digit, tscore, is incremented by 1 either when a one is carried from the ones digit or when the signal eat_pill is received. The hundreds digit, hscore, is incremented by 1 each time the signal eat_ghost is received or when a 1 is carried from the lower digits.

A look up table for each digit converts the number into an ASCII character. The characters are loaded into the output registers score_char on the negative edge of vsync so the characters string display does not receive new input when it is displaying an ASCII string.

2.4 Victory Animation

The victory animation occurs when the player has cleared a board. The red ghost, Blinky, chases Pac Man across the screen. Then, Pac Man chases a now blue Blinky back across the screen. The animation module takes as inputs the pixel clock, a reset signal, the negative edge of vsync, hcount, vcount, and the anim_start signal from the game controller FSM. Its outputs are the anim_done signal, and the pixels of the animation.

The animation modules creates a red ghost sprite and a Pac Man sprite. It has registers containing the x and y pixel coordinates of both sprites, a frame counter that controls the sprite animations, the direction that Pac Man is facing, and if the ghost is blue or not. The frame counter is a 6 bit counter that increments by 1 at each negative edge of vsync. Only the upper 3 bits of frame counter affect the animation frame of the sprites so the sprite animations are slow down. The animation module resets the locations of Pac Man and the ghost to their starting

locations when it receives the reset signal dots_reset from the game controller. The animation starts if the game controller FSM is in the victory starte, and the signal anim_start is high. The ghost chases Pac Man across the screen until Pac Man's coordinates reach the right side of screen. At that point, Pac Man flips direction, the ghost turns blue, and Pac Man begins chasing the ghost. The signal anim_done is activated when the ghost's x coordinate becomes 0.

3. Mike's Modules and Implementation

I was most responsible for the game logic. While Cassie generally worked on display stuff, I ensured that they were put to good use. I made it possible for all of the characters to move within the maze while obeying the physical constraints of the maze. Namely, I made it such that the characters wouldn't run through the walls. I also coordinated the animation of each character. Cassie made all of the frames of animation and I coordinated which frame was displayed. I was also responsible for all of the collision detection. Furthermore, I wrote the high level game logic; I controlled the flow of events while a user plays the game. I also wrote the four strategies for the ghosts. In addition to all of this, I set up the interface (PS2 Keyboard), wrote a soundboard for playing the numerous sound effects in the game and designed the title screen display. The title screen is the one bit of display work that I did.

3.1 Char_move.v

The character moving module generalizes movement for the five sprites in the Pac-Man world. This module was designed with the intent for working with both Pac-Man and the ghosts, even though there are slight nuances between the two groups of characters. In short, this module takes in the direction that a character wishes to go in, determines if they can go in that direction, animates their movement from point a to point b and then updates their coordinates. The actual execution of this idea is slightly more complex, and we shall get to that.

As one may recall from Cassie's explanation of the display, we tile set our map into 32x32 boxes so when we wish to go up, for example, we really want to move upward by 32 pixels into the tile above the character, but we shouldn't do that in one clock cycle (or one vertical retrace for that matter). We would like to smooth out that motion over a time on the order of a second so this module takes care of that. Throughout our project, we have this notion of a tile location and a pixel location. When we speak of a tile location, we are generally referring to some form of game logic and when we refer to pixel location, we are generally referring to some form of display.

This module takes in a clock, which was a 65mhz clock in our case, the global reset signal, the edge of a vertical retrace, the up, down, left, right and pause signals, a reset_x and reset_y coordinate, a character speed, a playing signal, a beign_sfx signal, an ai_done signal, a char_dead signal, a powerpill, a victory and a collision signal. This is a lot of inputs, but they are all essential.

Outputs include the character's x and y tile coordinate, their x and y pixel coordinates, their frame of animation, a ready signal, the character's current direction and passing information about where the character is located.

The clock input is necessary because we wish to synchronize our movements to a clock. We also take in the negative edge of a vertical retrace on the VGA. This is necessary because the char_move module is responsible for animating a character and we would like the animation to be smooth and pleasing to the human eye.

We take in the up, down, left, right and pause signals because these give us the information we need about the user's (whether it be a human or computer's) input. Without these signals, the move module would never do what it's supposed to do: move the character. The reset_x and reset_y are 5 bit registers that describe the tile location where we wish for the character to move to whenever we wish to reset their location. We may reset their location for a variety of reasons. In my implementation of moving a character, their location is reset whenever the global reset is asserted, whenever a collision is detected between Pac-Man and a ghost, whenever a level begins (begin_sfx asserted), or if the character ever dies (char_dead asserted).

The character speed is simply a 3 bit register that describes how long we wait before changing the character's frame of animation. While other people would use speed to describe how many pixels a character moves at a given time, I use the speed to describe how long we wait before moving a set number of pixels (which is four pixels in this case).

Playing is a signal is that asserted from the game logic and it tells us when we are playing the game, as opposed to being at the title screen or starting a level. We use this signal to enable all actions throughout the module. If we're not in the process of playing the game, characters shouldn't move.

Begin_sfx also comes from the game logic. This signal refers to the period in time where a level is starting up. This would be the time where some sound effect would be playing and it is the period just before one could play the game. In this module, it is used to prevent a character from moving.

The ai_done signal is given to us by the ghosts' individual AI modules. While the user can input the up, down, left and right signals at any time, the ghosts determine these signals by an algorithm described later on. We don't want the ghost to move until he makes a decision about where he wishes to move next so we use this signal as another form of enable on moving. Without this signal, we would run into some timing issues where the ghost would be continuously confused about his location and state and he would simply move back and forth between the same two locations. Since Pac-Man is not an AI, we hard wire a zero for this input when declaring Pac-Man's moving module.

The char_dead and powerpill signals are also used exclusive by the ghosts. When Pac-Man dies, the game logic will shift out of playing mode so all user input will be ignored. However, when a ghost is eaten, the game continues so we don't wish for a ghost to move if he is eaten so this is yet another enable that we use on the movement. We use the powerpill signal to ensure that

ghosts move at speed one while Pac-Man is Energized. Since we only want these two signals to apply to ghosts' movement, we hard wire two more zeroes in place of these inputs for Pac-Man's moving module.

Finally, we use the collision signal to tell the module when Pac-Man collides with a ghost and should die. In this case, we halt his movement. If we did not do this, then we would have some synchronization issues where Pac-Man would run into a wall upon being rest to his starting position. Since this signal only applies to Pac-Man, we hard wire zeroes for this input when declaring all of the ghost's moving modules. We use the victory signal for the same purposes because sometimes the player will actually clear the board and we don't want the same bug to happen.

Since this module is responsible for maintaining the character's position, we have it output the character's tile and pixel coordinates. The pixel coordinates are used for display purposes, and for collision detection. The tile coordinates are used to determine if Pac-Man eats a dot as well as determining the ghost's behavior. This information is also used to compute the possible directions that the character can move in.

The frame of animation output is assigned and sent to the character display controllers so they know which frame to display. This is used to animate the character as they move from square to square.

The ready signal is applicable only to the instances of char_move that are used to move ghosts. Even though it is assigned anyway for Pac-Man, nothing uses it. The ready signal tells the corresponding ghost AI module that the move module is ready for a new input. This is the AI's module to go ahead and compute the next direction that the ghost should take.

The direction and passing information are used primarily for the ghost modules. Essentially, we want the ghost to know what direction he was going and what directions he can go in. The direction output is a 2 bit register that describes the four possible directions that the character could have been going. Table 3.1, illustrated below shows the information that is represented by the direction register.

Table 3.1: Character's Direction

Direction Name	Symbol	Parameter Number
Right	D_RIGHT	00
Left	D_LEFT	01
Down	D_DOWN	10
Up	D_UP	11

The passing information is read from a ROM, which we'll refer to as a constraint ROM. This is represented as a four bit register. In essence, the ROM takes in an address, which is directly related to the character's x and y position and outputs a four bit number. Our map is 16x24 so the ROM has 16*24 = 384 entries.The MSB (bit 3) of this number tells whether the character can move up from their current location. Likewise bit 2 tells whether you can move down, bit 1 tells whether you can move left and the LSB, bit 0, tells whether you can move right from your current position. The bit is a zero if you can't go in that direction and a one if you can.

The moving module utilizes a simple finite state machine to control moving. Below is a summary of the control flow for moving.

- At rest, the module is in S_WAIT. In this state, the module is ready and waiting for a directional input.
- The user (or controller) sends a direction that the character wishes to go.
- The move module checks the information from the passing ROM to see if the character can, in fact, go in that direction.
- If the character can, the character's direction register is updated and the FSM transitions in the S_MOVING state.
- If the character cannot go in that direction, the input is ignored and the FSM stays in the S_WAIT state.
- When in the moving state, the character animates in its set direction. Every frame of animation corresponds to moving 4 pixels. As a result, there are 8 frames of animation. The amount of time between frames of animation is 8 vsyncs/speed. In the case where the speed is 1, the character changes frames every 8 vertical syncs and enumerates through 8 frames for a total of 64 vsyncs (approximately one second) to move from one tile to another. In the default settings, Pac-Man's speed is 4, Blinky's is 4, Pinky's is 3, Inky's is 3 and Clyde's is 2.
- While in the moving state, all input is ignored so a controller cannot try to change direction while we're moving from one tile to another.
- After going through all 8 frames, we have finished moving and the FSM transitions out of the moving state into the S_UPDATE state.
- In the update state, the character's x and y tile coordinate is updated dependent on their previous coordinate and their direction.
- After this state, we go into the S_DELAY state. This state is more or less a hack. The constraint ROM needs time to catch up and read the new directional information. I delay 4 clock cycles, but the ROM only has a pipeline latency of 1 clock cycle so the extra delay isn't really necessary, but added as a safety.
- We now know which directions the character can move so we transition into the S_WAIT state and wait for the next input.

Of course, all of this happens as long as all of the enable checks described above are met. For example, we must be in the playing state of the game; the character can't be dead, etc.

Now, one thing we will see later on is that the constraint ROM may have been placed needlessly in the char_move module. When we examine the ghost AI's, we will see that a ghost will only emit a signal (up, down, left, right) such that they go in a valid direction.

3.2 Cdetect.v

Now that we are capable of moving characters around, we want to determine when they collide with each other. While one would imagine that collision detection would be dependent on the character's x and y tile coordinate, I decided not to do this because then we didn't get very accurate results. Since the characters move at different speeds, it is quite possible for a character to collide with another while they're both in the moving state. Remember that a character's x and y tile coordinate is only updated AFTER the character completely moves into the next tile. For this reason, we use the character's x and y pixel coordinates to determine collisions.

In the appendix, my_x and my_y refer to Pac-Man's x and y pixel coordinates and all of the other pixel coordinates correspond to the individual ghost's x and y pixel coordinates. We also take in the powerpill signal as an input because while in power pill mode, we wish to detect collisions between Pac-Man and each individual ghost. Because of this desired behavior, we output not only a collision signal, but a signal for each individual ghost that tells them if they're dead or not.

In essence, we create a giant tree of logic that synchronously checks to see if Pac-Man's x or y coordinate matches each ghost's x or y coordinate and then checks to see if the unmatched coordinate overlaps with the width (or height) of the ghost. This is done synchronously, but only at the vertical retrace to lower the propagation delay. We don't need to check for a collision every clock cycle because a character changes x and y coordinates every few retraces (dependent on the character's speed). This keeps our data stable and our delay down a little. If we're not in powerpill mode and a collision is detected, then we signal collision so the game logic knows to decrement Pac-Man's life and start the level again (if he still has lives that is). If we are in powerpill mode, we know to tell the display and the moving module that the ghost is dead so he can be placed in the pen and not move, or be displayed until he respawns.

3.3 Pac_game.v

We're finally starting to get something here. We can draw characters, move them around (given that the user tells them where to go), and even eat the dots. We're still missing the actual game part. The game module serves as a wrapper for all of the other modules that we have written. It utilizes a FSM to control the game state and inform the other modules information about what state the game is in.

This module is clocked with the 65 MHz clock (much like all other modules). It also inputs the global reset. In addition, this module receives: start_button, collision, board_cleared, dots_reset_done, sfx_done, and anim_done. These signals primarily are used to trigger changes in the game state.

This module outputs: title_screen, reset_dots, begin_sfx, death_sfx, victory_anim, playing and a lives count. Most of these signals are used to signal other modules.

The FSM is fairly linear and illustrated below (on the next page).

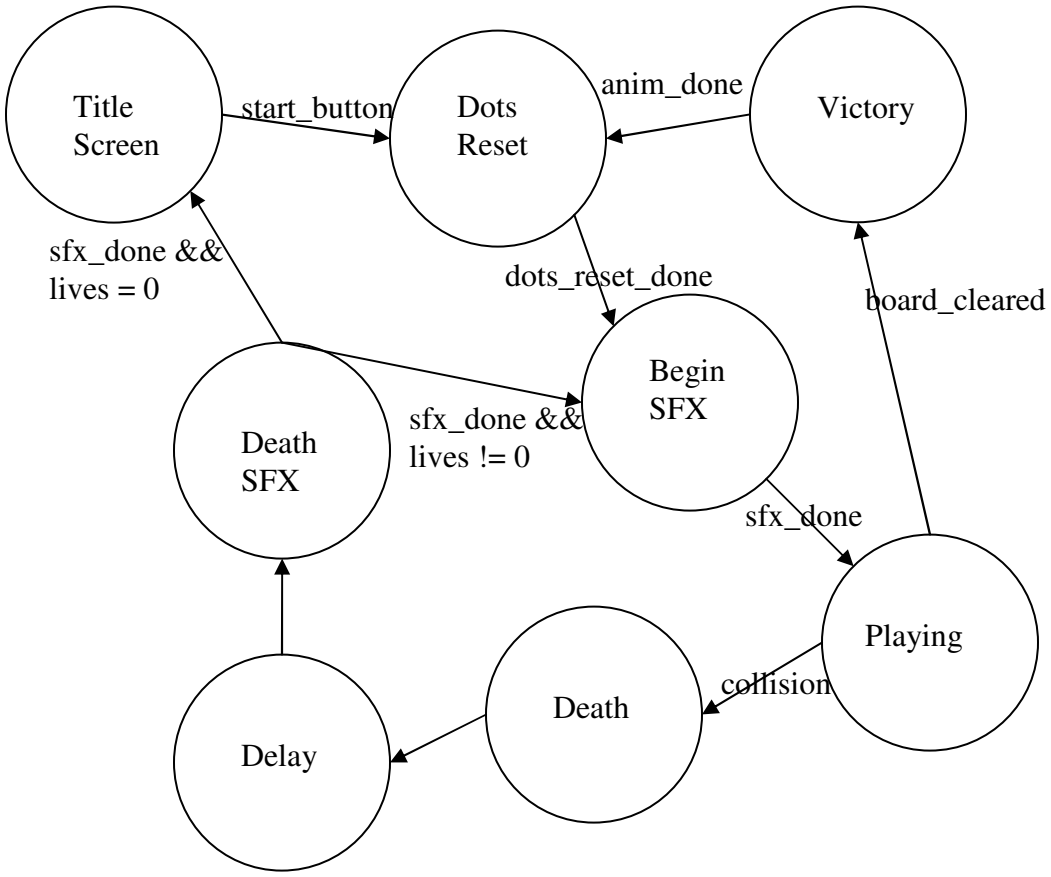


Figure 3.3 – Pac-Man FSM: This is the primary FSM for the game of Pac-Man.

Figure 3.3 above shows the FSM that represents the control flow for Pac-Man. The game FSM is responsible for maintaining the player's lives. The default lives that the game gives a player upon reset is given as a parameter, and subsequently can be redefined using the defparam command. Pac-Man is given lives at the title screen. Lives are only decremented while in the Death state. The only thing that may be a little misleading is that edge detection is done on the collision signal so that signal really transitions the game from the Playing state to the Death state. The FSM will output a signal corresponding to all states except for the Delay state.

The game starts at the Title Screen state. When the user presses the button, the maze will be drawn and dots will be drawn. The game will then go into the Begin SFX mode where the sprites will be places in their starting positions and ideally a sound is played to signify a level starting. In actuality, this state seems to pass through in one clock cycle. This occurs because the sound

module didn't work as intended. We shall get to that later on, but the state is still present. After the SFX is "done" we go into the main state of the game: the Playing state. This is the state where all of the interesting stuff happens. After all, this is the state where you can move around, eat the dots and get hit by ghosts. If Pac-Man ever gets the collision signal while in this state, signifying that he hit a ghost and wasn't Energized, then we transition to the Death state. In this state the lives are decremented by one and we go to a brief Delay cycle. The delay cycle is to ensure that the life counter contains the proper value. This ensures that we actually trigger a game over and go to the title screen if we lose all of our lives. We then go into the Delay SFX state where, ideally, we would play a sound effect and transition out. Once again, the bug in the sound module causes this state to transition out after one clock cycle. At this point we determine if we still have lives or not. If we do, we go back to the Begin SFX state and play the level again. In this state, the dots are not reset but all of the character's positions are. If we have run out of lives, we go back to the Title Screen.

3.4 Ghost AI

At this point, we have a game where you can pretty much do everything you would want to. The only problem is that the autonomous characters don't move autonomously yet. The game would not be very much fun if the player controlled all five sprites (which is how everything was tested up until this point). I then sought to create a scheme to move the ghosts in an interesting fashion.

In the Pac-Man game there are four ghosts. Their names are Blinky, Pinky, Inky and Clyde (as one may have discovered by this point). While it would be possible to make every ghost move randomly, this wouldn't be very interesting because a careful player could easily avoid all of the ghosts and gobble up all of the dots. Part of the charm of Pac-Man was that each ghost had its own personality. I took inspiration from the Pac-Man Wiki entry and sought to use them as a basis for creating my own ghosts with personalities. However, in order to do this, I created two generic strategies: wander and prioritized moves.

3.4.1 Ghost_ai_wander.v

In an effort to create a somewhat random strategy, I made set of rules to define what it means to wander. This module takes in the 65 MHz clock, a ready signal, the ghost's direction and their pass data. The wander strategy will use this information to determine the up, down, left and right signals to output to the higher ghost AI module (which will then, in turn go to the moving module).

Since there is pseudo-randomness to this module, there is a 32-bit Linear Feedback Shift Register. This system was created via the IPCoregen tool chain. What the system creates is a series of shift registers all fed back into a 32-bit XOR gate. The data was seeded synchronously with the hex value ABCDE and the entire number was read out in parallel. We then picked a bit, designed by the concatenation of the bits 0, 1, 5, 7, and 3 from the 32 bit "random" number. I could have just as easily picked the LSB or the MSB, but I figured to be creative on choosing which bit to use.

In our maze, all squares that a ghost can exist in are either two directional, three directional or four directional. The lone exception is the narrow 1x2 channel at the lip of the ghost pen (which is one directional), but we will see that the ghosts will have logic to handle the case where they are in the ghost pen.

If the ghost is in a two directional pathway, they will simply continue along their way. If, for some reason, the ghost wasn't traveling in either direction, he will pick a random direction and stick with it. This happens when the ghost first leaves the ghost pen. He is traveling up, but the square he lands in only allows him to move left or right. The extra rule serves as a failsafe to ensure that the ghost doesn't get stuck (at least not at a two directional pathway).

If the ghost is in a three directional pathway, then they will use the random bit to decide which pathway to take. They will choose between the paths such that they only consider the ones that keep them from going backwards. We don't use the failsafe explained above because I reasoned that the ghost could never be stuck at a three way intersection. The only time a ghost could be stuck is if he is coming out of a one way passageway and can no longer continue in his previous direction. Let's briefly examine the case where the ghost goes up from the ghost pen. If there were a three way passageway, he would have to be able to go right, left and up. He can't go down because he came from a one way passageway. In this case, the ghost would simply continue going up until the next intersection. This also assumes that there are no dead end pathways, which is quite the reasonable assumption for Pac-Man.

Lastly, if the ghost is at a four directional intersection, they will simply do nothing and maintain their direction (much like the two direction case). In the case where the ghost reaches a corner, he will take the corner and continue along his merry way.

While this scheme is fairly simple to explain, it was very tedious to make. Under this strategy, the ghost knows the direction that he was traveling in and he knows which directions he can go in. Therefore I had to enumerate all of the permutations of his possible direction and his previous direction. This logic is only valid when the ready signal is asserted. This means that the ghost only makes a decision when the char_move module tells the ghost module that it is ready for new input. This keeps the ghost from running past his mark and becoming confused about his location.

3.4.2 Ghost_ai_precmove.v

Other than a strategy that just wanders aimlessly, I sought to create a more powerful strategy. After some careful thought, I realized that I could generalize a lot of strategies to the idea of prioritized moves (or moves with precedence as I initially called them). This module accepts the 65 MHz clock, the ready signal, the ghost's pass information and a sequence of precedent moves. With this information, the module outputs the up, down, left and right signals.

Everything is as before, except for the removal of the ghost's direction and the addition of this prec_move register. This register is 8 bits wide and contains a sequence corresponding to the

directions that the ghost wishes to go in. Every set of two bits corresponds to a direction, as described in Table 1 above. The top two bits correspond to the move with highest priority. The next two bits correspond to the direction with second highest priority. The 3rd and 2nd bits correspond to the direction with third highest priority and the lowest two bits correspond to the move with lowest priority.

The goal of this strategy is to move the ghost in the direction with highest priority possible. The module knows the order of moves that the ghost wishes to go in and it also knows the directions that the ghost could move in. Using these two pieces of information, it is fairly easy to design logic to move the ghost in a valid direction with the highest priority possible. This was accomplished with a fairly complex set of case logic wherein all possible sets of valid directions were enumerated and all possible precedent moves were enumerated. This was done for all sets of two, three and four way intersections. Consistent with the wander strategy presented above, this logic is only valid for when the ready signal is asserted.

We will see the power of this prioritized move strategy when it comes to generating the behaviors of each individual ghost.

3.4.3 Clyde_ai.v

Clyde is the green ghost that moves erratically. For this reason, we have him generally wander. This module takes in vclock, reset, playing, ready, clyde_x, clyde_y, dir, pass, and dead as inputs and gives up, down, left, right and ready_done as outputs.

The clock is our 65 MHz clock as usual. The ready signal still tells us whether or not to generate our next move. The rest signal resets all outputs to zero. We use dead and playing as enables on our logic. We don't want the AI to generate moving signals if the ghost is either dead or if we're not playing. The pass and dir inputs are sent to the wander module and are used as described above. We use the ghost's x and y coordinates in order to get him out of the ghost pen. Since the ghost pen is a one directional pathway, we do not wish to break our system because it doesn't know how to handle one directional pathways. Therefore we hard code all coordinates that he could have while in the ghost pen and tell him explicitly how to get out. In the case for Clyde, he must move left first and then up twice.

If Clyde is not in the ghost pen, then we simply output the up, down, left and right signals as given in the wander module. While most other ghosts will adopt a separate strategy when Pac-Man is Energized, we allow Clyde to be the stupidest form of a ghost and continue to wander aimlessly while he is helplessly gobbled up by Pac-Man. While his actions are irrespective of Pac-Man's actions, this ghost can be the trickiest to avoid because his moves are fairly unpredictable. Of course, the ghost will repeat his actions after a set period of time, but a higher degree of randomness is introduced when the player dies. The shift register is seeded only on reset. This means that every time after the reset button is pressed, the ghost will move in the same pattern as he always does after the reset button is pressed. After watching this for a while, his movements become quite predictable (believe me, I would know). However, after a player has played for a while and dies, the random number generator isn't reset; it will continue

generating the same bits as it always has. However, what has changed is the ghost's location relative to the bits that the shift register is outputting. This means that the ghost will run more random patterns than expected, unless the player happens to die at the exact time that the generator begins to repeat itself.

Finally, the ready_done signal is asserted and sent to the move module to let it know that the AI is done computing its next move. In this case (and the case of all the ghost AI modules), the ready_done signal is merely a hack where was delay for a "magic number" of clock cycles. In this case, I picked that magic number to be four clock cycles and it behaved well. This may be a slight inefficiency in my logic, as we may not need to wait that long, but it's a safe delay.

3.4.4 Blinky_ai.v and Pinky_ai.v

Blinky's AI module is quite different from Clyde's. However, it is quite symmetric to Pinky's AI. It accepts all of the same inputs as Clyde's (naturally, it receives it's own x and y coordinates instead of Clyde's), except this ghost also receives the power pill signal and Pac-Man's x and y coordinates.

The goal for Blinky is to actively chase Pac-Man. We accomplish this by using the idea of a prioritized move. If we are not in powerpill mode, then the ghost wants to chase Pac-Man. If we are in powerpill mode, we want the ghost to run away from Pac-Man. Pinky follows the same rule set. However, each ghost chases Pac-Man in a different fashion.

For Blinky, we have him examine his x and y coordinates and Pac-Man's x and y coordinates. Depending on the position of Pac-Man with respect to Blinky, he prioritizes his movements to approach Pac-Man in a clockwise fashion. Figure 3.4.4a below shows how Blinky would go about approaching Pac-Man given that Pac-Man is to Blinky's lower right.

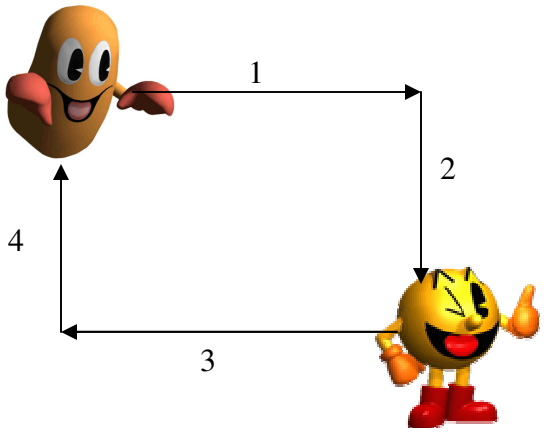


Figure 3.4.4a – Blinky's Chase Strategy: This illustration shows how Blinky would precedent chasing Pac-Man if Pac-Man is to Blinky's lower right.

If we prioritize our movements in this manner, then we will cause Blinky to always move towards Pac-Man in a clockwise manner. For Blinky's AI, we have him wander if Pac-Man's x coordinate is more than 6 away from Blinky or if Pac-Man's y coordinate is more than 10 away. This makes the ghost less ruthless in chasing Pac-Man. If Pac-Man is within this range, we have the ghost chase him using the strategy outlined above.

For Pinky, we do something very similar. The only difference is that Pinky now approaches Pac-Man in a counter-clockwise fashion. Figure 3.4.4b below illustrates this idea.

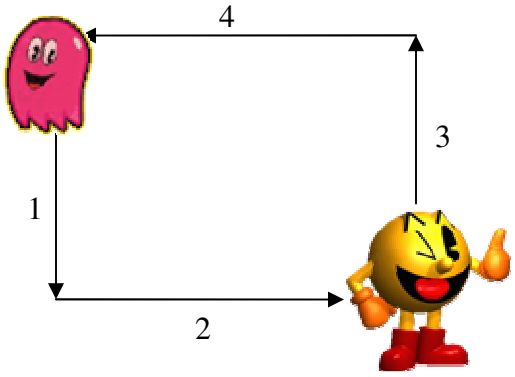


Figure 3.4.4b – Pinky's Chase Strategy: This illustration shows how Pinky would precedent chasing Pac-Man if Pac-Man is to Pinky's lower right.

The neat thing about this set of strategies is that combined, we have a set of ghosts that will effectively trap Pac-Man. In testing, this simple, yet powerful strategy caused many frustrations for the player, as the ghosts would trap the player if they got too close. By using a wander strategy, we allowed for the ghosts to have a false sense of being harmless. Also, without mixing the two strategies, the ghosts would sometimes get stuck because they would be unsure as to how to get to Pac-Man. This occurred if Pac-Man was too far away from the ghosts or near the 'S' patterns that the walls make near the sides of the middle of the maze. Therefore, I made them wander a bit just to jump start them again so they wouldn't continuously move back and forth between the same two spots. This is the one aspect of the project that I am most proud of. Although it appears to be too simplistic in explanation, it is a very effective strategy that allowed for much better gameplay. The player now has to think harder about the directions that he wishes to take. Of course, a really smart player could beat the ghosts every time because their moves are mostly deterministic, but this system is far superior to the generic random movement.

If the ghosts see that we are in powerpill mode, they will run away from Pac-Man. We will explain how this is done in the next section, because Inky's strategy is to run away all the time.

3.4.5 Inky_ai.v

Our last ghost, Inky, is the bashful one. He loves to run away. His AI setup is very much like Blinky and Pinky's. He gets all of the same inputs and gives the same outputs as them. What is different is how Inky precedents his moves. We precedent Inky to move counter-clockwise AWAY from Pac-Man. We are still using the same module that moves the ghost in a prioritized fashion. Figure 3.4.5 illustrates how Inky would move given that Pac-Man is to his lower right. We should note that prioritizing our moves clockwise away from Pac-Man will still move the ghost away from Pac-Man. In this case, Blinky will move clockwise away from Pac-Man and Pinky will move counter-clockwise away from Pac-Man.

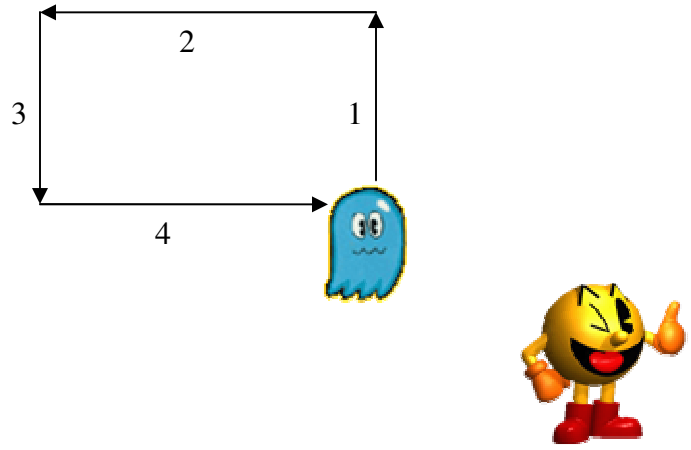


Figure 3.4.5 – Inky's Run Away Strategy: This illustration shows how Inky would move, given that Pac-Man is to his lower right. In short, Inky will run away.

There isn't much benefit to assigning priority to clockwise over counter-clockwise when we wish for the ghost to run away. The only thing I can really think of is that the ghosts will split up and force Pac-Man to choose which one he wants to eat. Since all of the ghosts are the same color, the player may not remember which one is the speedy, red ghost and they may eat Inky by accident (who is by most means harmless).

In order to prevent monotony, we have Inky mix between running away and just generally wandering. We set him to wander if Pac-Man's x coordinate is more than 8 away or if his y coordinate is more than 12 away. Essentially, Pac-Man must be on the other half of the board; otherwise, Inky will run away. This implies that Inky will tend to stay either at the top or the bottom of the board.

3.5 Keyboard Interface

For the better part of this project, we used the labkit buttons to control moving Pac-Man up, down, left and right. While this wasn't horrible, it did become a pain, literally. The buttons on the labkit definitely hurt one's fingers after a while. We had initially intended to interface with a PS2 controller, but we never quite got around to it. We didn't think it was a huge part of our design. Since Professor Chris Terman wrote a module that would handle keyboard input, I figured that it would be silly not to put it to good use. On the top level labkit module, I declared an instance of Prof. Terman's module and wrote the logic to handle what would happen when the W, A, D, S, enter or space bar was pressed. We left the reset button to still be wired to the labkit's enter button because it was a global reset value (almost like the reset button on a gaming console). In essence, I made it such that whenever one of those buttons were pressed, the corresponding signal (up, down, left, right, start and pause) and only that corresponding signal was asserted. This caused an accidental behavior that was actually desirable. In the original Pac-Man, the controls were such that Pac-Man goes in a direction and keeps his direction until you change it. When we simply used the labkit buttons, one would have to hold down the button to move. Now, one only has to press the button when they wish to change direction. The behavior is more true to the original and our fingers are spared.

3.6 Soundboard.v and play_sound.v

The soundboard was made initially as an external system. Its intention was to play a given sound when triggered to. The sound files were actually WAVE files that were first sampled to 12khz, read in Matlab to attain amplitude information and then converted to be a number between 0 and 256 (8-bit data). The converted data was saved as the .coe file and read into a ROM in Xilinx through the standard IPCoregen process. In my soundboard, there were 5 possible sounds: begin.wav, ghosteaten.wav, interm.wav, siren.wav and killed.wav. Begin.wav was the sound that corresponded to the startup music that plays when a player starts a level. Ghosteaten.wav is the sound that is played when the game is in powerpill mode and a ghost is eaten. Interm.wav is the short interlude that plays when the player clears the board and the victory animation is displayed. Siren.wav is the looping background sound that a player hears when the ghosts are active and

can eat Pac-Man. Lastly, killed.wav is the sound that plays when Pac-Man is defeated and eaten by the ghosts.

Another module, play_sound.v, was written to play an individual sound. The sound samples are interpolated linearly for a better playback. The code for this module was inspired heavily by the recorder/playback system that we developed earlier in the semester. It also utilizes the audio and audiocommands modules that were given to us earlier in the semester. These two modules interface with the AC97 codec onboard the labkit. Linear interpolation was achieved by using two consecutive samples as our endpoints and then finding the equation of the line between the two points. Since the two samples are four time units apart, we can easily draw a line at even intervals such that we step up or step down to the desired value.

When the module was independently tested and the sounds were triggered by the switches, all worked fine. However, when this system was integrated, the sounds would never play. I feel as if I know the cause of this bug, but I have lacked the time to fix and test it. The soundboard.v module accepts the 27 MHz system clock, the reset signal, the ready signal (which corresponds to when the AC97 codec is ready for a new sample) and all the individual signals that correspond to playing each .wav file. The module outputs a signal corresponding to when it finishes playing a particular sound. In my implementation, I set those signals to be asserted when the address was zero instead of when it was equal to the high address for that sound (which I stored in a parameter because I knew them beforehand). This would definitely explain why the Pac-Man game FSM would never play the sound and transition out of the state where it should; the done signal for that sound file in the soundboard is asserted before the sound is even played.

3.7 Title Screen

The title screen was the absolute last thing that I worked on. The title screen was made in a graphics program (Photoshop). I saved it as a bmp file. I had initially intended to use Matlab to attain the 8-bit, RGB data from the image and write that to a .coe file, but I learned that Professor Ike Chuang wrote a neat python script that would convert a .pgm file to a .coe file. While there would be color loss, it proved to be infinitely easier (and quicker) to sacrifice the color loss to get a title screen into our game. The title screen image was 256x192 (each dimension of the screen was divided by 4). We converted this image to a .pgm file and used the python script to convert it to a .coe file. I then made a ROM with this image in the top level labkit module. There was one slight problem: all of our colors from the game were 3 bit and the title screen has 8 bit color. I remedied this problem by hacking the VGA outputs directly. If we were in the title screen, then the VGA output would be solely the title screen pixels and it would be the Pac-Man game pixels otherwise. This resulted in a nice final touch to our project. We had a better looking title screen, with our names, our project's name and the controls for the game. We had created for a nice way for the user to know how to play the game without having to ask us how the system works. This is ideal from a user's point of view.

4. Testing and Debugging

4.1 Cassie's Testing and Debugging

Testing the display modules required synthesizing the project, and then viewing the project on the screen to find any glitchy video outputs. When testing to see if a signal was generated properly, that signal was hooked up to the LED output on the labkit. This method was mainly used to test the signals generated by Pac Man eating a power pill.

4.1.1 MAP DISPLAY

The first iteration of the map display was a collection of 38 blobs connected to the output by an or gate. This worked as long as no further modules were connected to the pixel output, and the screen started showing glitches as soon as the dots were displayed. The glitches were caused by the propagation delay of the large or gate, and to cut down on the amount of logic, the map was read from ROM.

The second problem that arose with the map display was that the display was off by one line. The first row of the map was displayed on the second row on screen. This was caused by the one cycle delay between when data was read from ROM and when it was displayed. Originally, the address to the map ROM was set to 0 when hcount and vcount were both equal to 0. Changing the reset values of hcount and vcount to 1343 and 805 respectively fixed this problem.

The last problem with the map display was that there was a line of glitchy output every 32 lines. This was caused by the address changing every clock cycle that vcount was a multiple of 32 instead of only once. When the line was displayed, the address changed for every pixel as the module tried to display the entire rom during that one line. This bug was fixed by telling the ROM address to only change once when vcount was a multiple of 32. Instead of the address increasing by one if vcount was a multiple of 32, it now only increased by 1 at the end of a row of pixels where vcount was a multiple of 32.

4.1.2 DOTS DISPLAY

The initial dots display module had each dot as a separate blob. To populate the map required a total of 308 blobs, and a corresponding large or gate connecting those blobs to the pixel output. The propagation delay caused by an or gate of that size caused all the dots to be very glitchy. Changing the display from XVGA to VGA fixed the glitchy dots temporarily, but did not get rid of the very large or gate. To reduce the amount of logic needed to display the dots, the dots were read from memory and displayed in a manner similar to that of the map.

Several more bugs appeared when the dots display was modified to handle Pac Man eating dots, and dots being refreshed from ROM. The first error to appear was that Pac Man would eat an

entire row of dots at once. This particular error was caused by the write enable signal to the RAM being high before there was valid data in the write port. The write port didn't have time to load the last set of data read before the write enable went high. The data written to that address was a line of zeros. This error was fixed by not setting write enable to high until the write port had valid data loaded.

Another bug occurred when the data from the ROM was being written to RAM and then displayed. The new data written to RAM was off by one line. This was most likely caused by the address to the ROM and the RAM being the same. There is cycle delay between the time the data is read from ROM and when the data is written to RAM. The data from row 1 of the ROM would be written to RAM when the address pointed to row 2, so the new dots display was off by one line. This was fixed by having the ROM address be one higher than the RAM address.

4.1.3 PAC MAN SPRITE

It took several iterations of manipulating the ROM address to display the Pac Man sprite. Generally, a row was being read too many times, or the wrong bit of a row was selected as the output. More bugs appeared when the Pac Man sprite was configured to change which direction it faces. A shift of three had to be added to the x offset when selecting the row to read from when the sprite is facing up or down. This is caused by delay between when the address is updated, and when it is received by the ROM. This delay was not noticed when the sprite faced left or right since the address did not need to change every clock cycle. The delay became evident when the address had to change every clock cycle.

4.1.4 GHOST SPRITE

The ghosts sprites experienced no glitchiness when they were first implemented, but as more modules were added to the project, ghosts would randomly become glitchy. This is most likely caused by signal paths with long propagation delays. This was fixed somewhat by pipelining the pixel output. However, glitchy ghosts still appear occasionally. The current version of the project is fine tuned so the ghosts do not glitch. These could probably be permanently fixed by not displaying the video in real time, but instead storing the whole screen to a RAM, and reading from that.

4.1.5 LIVES DISPLAY

This module was simple enough that no errors were found during testing.

4.1.6 POWER PILLS

The first module written and tested was the power pills display module. At first, Pac Man would either not eat the pill as he entered that tile, or he would eat the pill when he was still a tile away

from the power pill. This was caused by having incorrect tile coordinates for the power pills, and was easily fixed by changing the tile coordinates of the pills.

The second major bug was the collisions were not detecting correctly. First, ghosts in powerup mode would still eat Pac Man. On the second trial, Pac Man continued to eat the ghosts when the powerup mode had expired. The last case involved Pac Man and the ghost mutually killing each other when they collided. Since all these errors involved collisions, I suspected the fault lay in the logic of the collision detector. After I cleared up the mismatched begin and end statements, and added correct logic to handle the powerup mode, the collisions were detected correctly. The ghosts ate Pac Man when they collided when powerup was off, but Pac Man ate the ghosts when powerup was on.

Another bug that occurred was that the ghosts would be eaten, and would never reappear on the screen. This was caused by the timer for the dead ghost module to be set to expire at 8 seconds while the output was told to reset to 0 when the counter reached 16 seconds. The ghosts reappeared after the given time after the timer was set to expire after the correct length of time.

One rather perplexing bug was that the ghosts would respawn in random locations on the screen, zip around, and finally end up inside the walls of the ghost pen. This was caused by the movement and ghost AI modules sending the ghost movement information when the ghost was dead. Rewriting the modules to include ghost_dead as an additional case where the module was set to reset mode fixed this problem.

The last bug with the power pills came when the player cleared the board. In the prior implementation, the power up counter and the dead ghost counters only reset when the reset button was pressed instead of every time the screen refreshed. This caused the effects of power pills to carry to over after the board was cleared. Dead ghosts were still dead, and power up mode continued even when the screen reset. This was fixed by wiring the reset input of the counters to the dots_reset signal from the game controller FSM instead of to the reset button.

4.1.7 SCORING

For several weeks, every time the project was synthesized, a warning popped up saying that there were three gated clocks in the scoring module. A closer look at the module revealed that there were three case statements that did not have enough entries to cover all the cases, and no default statements, thus causing Xilinx to think that they were clocks. Adding default statements to the case statements fixed this problem, and decreased the amount of logic used. No other bugs were found in the scoring module during testing.

4.1.8 VICTORY ANIMATION

The first bug in the animation came when the sprites would chase each other from left to right across the screen, but get stuck trying to chase each other from right to left. This was caused by the contents if statement that was true if the x coordinate of Pac Man was a certain number.

However, the contents of that if statement never changed the Pac Man's x coordinate, so that every clock cycle thereafter, the if statement was true, and the logic would never move to the else statement. This was fixed by changing Pac Man's x coordinate in the if statement.

The second bug came at the end of the animation. When the animation ended, the screen would refresh, and Pac Man would move in the last direction he was traveling and get stuck inside a wall. This bug was bypassed by having the game controller go to the title screen after the animation finished instead of refreshing the screen immediately. This bug can probably be fixed by adding a new input to the movement modules that tells the character to stop moving during an animation.

4.2 Mike's Testing and Debugging

Since our project revolved primarily on VGA display, the general form of testing was to synthesize the project and run it and watch the VGA screen. If things did what they were supposed to, then it would be clearly evident from the display. In order to help the process of debugging, various signals would be brought out and displayed on the labkit's 16 digit hex display and the 8 LEDs.

4.2.1 Char_move.v

This was the first module that I wrote. It was also the most painful one to write. This is evident as I spent approximately three pages explaining how moving works. It took me nearly a week to figure out the exact rules for moving a character. Part of the problem stemmed because, even though we had a block diagram, one would not expect it to be perfect. Our project had no real substance in the beginning. Therefore, a lot of our ideas were very abstract so it was difficult to conceptualize what would happen once everything was working. It took a long time for me to come up with a scheme that would smoothly animate the character. There were also many synchronization issues wherein the character's x and y coordinates wouldn't be updated properly nor would they be displayed in the proper position. This was remedied by spending lots of time analyzing my output signals via the hex display and LED display and thinking about my desired behavior. It also helped to jot down many notes on scrap pieces of paper and draw the little FSM that was the moving module. By doing things this way, I was able to get the base module that worked. As the project became more complex, I constantly had to modify this module because there were various other checks that were performed (such as the AI being finished computing its value before allowing the character to move).

Another problem that occurred consistently throughout designing this module was that the characters wouldn't move smoothly. It really wasn't that difficult to get the character to move the whole tile every time a button was pressed and it wasn't too difficult to get the constraint ROM working. The real challenge was to get the animations to be smooth. With some tinkering, I was able to come up with a scheme for counting vertical syncs. This was actually done by having a 6 bit counter that would increment by the char_speed value every vertical retrace. The frame count

would simply be the higher order three bits of this counter. Initially, I tried using two counters, but I ran into synchronization issues because the frame of animation was being assigned synchronously so there was a delay associated with assigning the frame of animation and actually displaying the frame of animation. This caused for an ugly display. After finally realizing that the frame of animation should be assigned through combinational logic, then the move module animated as it should.

4.2.2 Cdetect.v

The collision detection module was fairly straightforward. However, I did some amount of testing to allow for the best behavior possible. Initially, I had it set such that it would check to see if Pac-Man's x coordinate lined up with the ghost's x coordinate and then used the opposite pixel coordinate to determine overlap. Note that the final version uses purely pixel coordinates. When I used tile coordinates as described above, the ghosts would be able to easily catch Pac-Man while turning corners. It is true that if Pac-Man is turning a corner while being chased by a ghost, he will overlap the ghost at some point. However, we don't want our game to be overly cruel and say that this is a collision. Therefore, I changed it to check for pixel coordinates. As a result, I ensured that the coordinates would have to be stable for some time before a collision is detected. For example, assume that Pac-Man is at a corner such that he can only go up or left and a ghost is to his immediate left. If the ghost goes right to eat Pac-Man and Pac-Man goes up at the same time, there will be some overlap. However, no collision will be detected because Pac-Man will be in the process of leaving the tile. This implies that collisions only occur at corners if Pac-Man is caught completely in the tile when the ghost overlaps him. This scheme works nicely because Pac-Man can still be caught if the ghost fully enters the tile before Pac-Man fully leaves the tile (in this case, the x coordinates of Pac-Man and the ghost would be equal and their y coordinates would overlap instead of the opposite situation occurring).

4.2.3 Pac_game.v

This was also a fairly straightforward module. Since it is the Major FSM of the system, I simply outputted the state to the 16 digit hex display and wired the inputs that I didn't have automatically generated (dots_reset_done, sfx_done, and all other inputs that triggered state changes) to the labkit buttons. This allowed for me to test this system independently from the project. Once I was satisfied that it worked as it should, then I hooked in the signals that I already had generated via the Pac-Man game. In this case, it was just the collision detection signal because we had not yet made dots work. I then made sure that the game would cause you to lose a life by colliding with a ghost. Once this worked, I was happy and moved on.

4.2.4 Ghost AI

When I made the ghost AI, I initially thought that the random strategy (the wander strategy) would be the easier of the two to make so I made that one first. There were actually a lot of synchronization issues between the ghost AI and the moving module. Even though I wrote logic

that appeared as if it would allow for the ghost to move as described above in the wander strategy, the ghost wouldn't do that. He would double back and generally just alternate between the same two squares. He would move a little, but not get very far because he would be moving back and forth between the same two squares most of the time. In essence, he was doing a random walk. A random walk was too random for our game so we had to tone it down a bit. From the module description given above, it should be apparent that there is a two way line of communication between the char_move module and each ghost AI module. Initially, there was only a one way line of communication; namely, the char_move would only tell the ghost AI when it wants a new directional value. Since the move module didn't wait for the output to be generated, it would use the old value to move the ghost. For example, let's assume that the ghost is moving up and he reaches a three-way intersection where he can go up, down or right. He decides to move right, but because of the logic error, the ghost will continue to move up. He now reaches the next square and wants to move right. However, let's assume that he can only go up or down in this situation. The rules say that if the ghost is in a two way section and isn't going in either direction, he will just pick a random direction. Therefore, the ghost would sometimes double back and encounter the same problem again. It took a while to realize that we needed that second line of communication between the two modules. However, once the bug was realized, it was easily remedied. I discovered the bug by wiring the up, down, left and right signals for that ghost to the LED buttons and pausing the game as the ghost reached a three way intersection. I witnessed that the ghost AI was sending out a signal to move to the right, but the ghost was, in fact, moving up. This helped me to converge on the error in my logic.

After changing my design to allow for this bi-directional pathway of communication, it was fairly simple to write the rest of the ghost modules. The prioritized move required no fussing with shift registers so it was fairly easy to generate the lookup table to give the desired behavior. I was convinced that the other ghosts' AI systems were good when I was able to place myself in a situation where the ghosts would, indeed, corner me and win. The same holds true for Inky's AI module. I was convinced that his module worked when he ran away from Pac-Man. Once again, we used actual game situations to determine if a system worked as intended.

4.2.5 Keyboard Interface

Luckily, most of the keyboard interface was already written for me so testing was fairly minimal. Testing was done by simply playing the game. If Pac-Man didn't move as intended, then I had to track down the source of the bug and fix it. The predominant problem was that after dying, one would immediately run through a wall near the starting position (if the player's last move was up or down). This was fixed by forcing the values for up, down, left and right to zero upon a collision. A similar fix was employed in the char_move module. For that module, the player's frame count was forced to zero if a collision was detected and the state was forced to the wait state (this is the only time a move can be interrupted).

4.2.6 Soundboard

The soundboard was first generated as a separate project. It was debugged by using the switches on the labkit to play each sound. While this did work, integration with the Pac-Man game did not. A copy of the working soundboard has been included with the project file to be perused at one's leisure. The code is slightly redundant, because it has been included in the Pac-Man project. However, the difference is that the separate soundboard project actually works. As stated previously, part of the reason why integration failed was because of faulty logic related to when the sound was finished playing and the lack of enough block ROMs on the labkit to actually generate the sound ROMs necessary for playing the sound files. The soundboard was made and tested by listening to the sounds. When I had aural verification that the sounds were playing properly (or somewhat close to properly), then the module was declared to work (or at least somewhat work).

4.2.7 Title Screen

The title screen actually was quite straightforward once I had the .coe file generated. The only issue was generating the address. I had to remember that I needed to repeat each pixel four times so I had to throw away the bottom two bits of hcount and vcount in the XGA display. I initially forgot to do this so I ended up with 12 copies of the image on the screen. A simple smack to the forehead and a small change to the code caused this problem to vanish into thin air.

5. Conclusion

5.1 Cassie's Conclusion

I learned several important concepts from this project. The first concept is that there's always a simpler way to do something than brute forcing it with lots of logic gates. A 300 input logic gate means that I'm probably doing something wrong, and I should change my approach. I also learned that when integrating code, I should integrate in small portions, testing as I go along instead of integrating all at once, having the newly integrated code crash, and no knowing which portion failed to integrate properly.

If I had more time for this project, I would first change the video display to be read from RAM instead of having a real time video display. Reading from RAM would likely cut down on the number of glitches caused by propagation delay. Also, I would add in additional levels to the game

5.2 Mike's Conclusion

Our final project enabled us to create a fully functional Pac-Man game. While our design may not be the most efficient one, it is one that most definitely works and works within the resource constraints that we have been given.

There are a few things that I have learned and would do differently if I had the chance to re-implement this project. For example, the collision detection module appears to be more complex than it needs to be. The logic can definitely be condensed into something slightly more efficient. However, I never got around to doing that due to the time constraints that haunted us all.

I would have also liked to develop more ghost AI systems. I did learn that sometimes the simplest system works the best, but I would have liked to make more simple systems. I had initially intended to make some complex ghost AI that would run set pathways given the state of the game and the positions of the characters on the screen, but I never actually did that. This idea was forgotten about once I realized the effectiveness of the simple strategy of closing in on Pac-Man from two different directions. I definitely learned that it is much more desirable to have something simple that works really well as opposed to something complex that doesn't.

I also learned the vast importance of communication. Most other groups performed projects that were really two separate projects that ended up being integrated together at the very end. Cassie and I worked on similar aspects of the same project. Therefore, it was very important that we constantly communicated with each other. We were constantly integrating our code and then diverging again to work on parallel tasks. When we met to integrate, we would have to communicate our changes to each other, how they behaved and how to use the modules that we had written. We probably should have used some form of version control, such as CVS, but we ended up doing things completely verbally. This worked because we only had a team of two people. Had we had more people, then we would have probably used version control. A big part of the success of our project is attributed to our ability to communicate effectively to each other.

Overall, I am quite happy with the modules that I had written. They are all fairly modular so it wouldn't be too difficult to do things like add more ghosts, add another map and add a second player for simultaneous Pac-Man play.

It took eight people fifteen months to build Pac-Man from scratch in 1980. A team of two sought to recreate this classic in five weeks. They were successful.