ECE 385

Spring 2024

Final Project
Pac-Man

Maxwell Krieger and Nicholas Costello
XC /4:15-4:30
Shixin Chen

Introduction

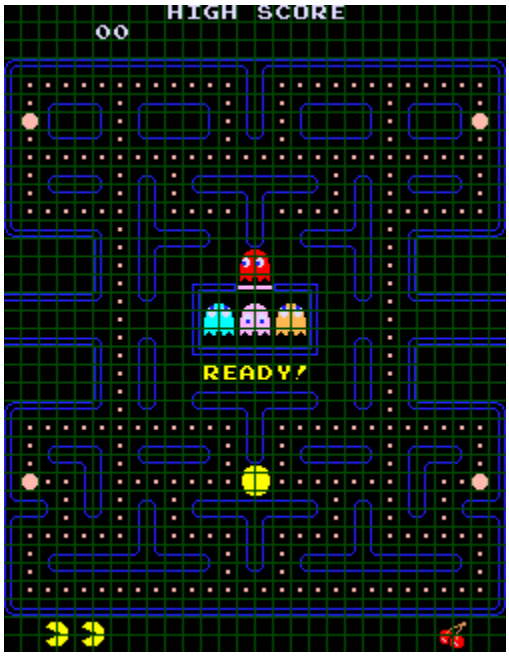
In our final project we designed Pac-Man. We were able to achieve this through a Microblaze processor, GPIO peripherals which helped handle keyboard input, and BRAM for additional memory storage as we had more intensive graphical features. Keyboard inputs were handled by utilizing the USB protocol of the MAX3241E Chip to allow for players to use a keyboard to control the movement of Pac-Man. The functionality of our game consists of Pac-Man traversing the maze with three lives to try and eat all of the pellets. His opposition, the three ghosts, however try to stop him by chasing and then eating Pac-Man in turn losing a life. The ghosts try to take away all of Pac-Man's lives before he can eat all of the pellets, in turn allowing them to win. Pac-Man however can play some offense as well by eating any of the four power-pellets on the screen. Eating one of these allows for Pac-Man to now eat the ghosts for a five second period, and lock them back into their starting point until time is up. Each pellet Pac-Man eats is 2 points, once he eats all the pellets (480 points) without losing all of his lives he (and the user) have won the game.

Game Design, Features, and Logic

Background

Most of our game logic was supported by our maze background as it allowed us to know where everything was located on the board. The maze background was constructed of 8x8 tiles with the board size being 28x36 tiles. With the vga displaying 80x60 tiles we centered the board on the screen, and blacked out the rest. To actually implement the maze background we designed the tiles ourselves by making our own FontROM. This background fontrom consisted of 45 different tiles which were unique to the background. This included the maze walls, pellets, the letters to spell SCORE, and the blank background. Because each tile only consisted of two colors

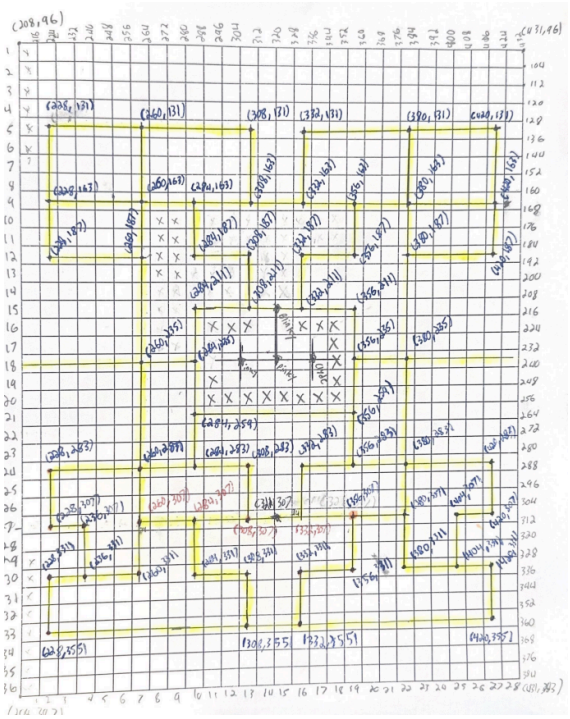
we were able to design each tile using 1's and 0's. To display the correct color and tile, we stored this information into BRAM. In BRAM we stored 9-bit codes. The first three bits [8:6] of this code determined which color would be chosen if a 1 was read from the fontrom. 3'b000 represented blue, 3'b001 represented beige, 3'b010 represented white, and 3'b011 represented purple. The last six bits [5:0] held the code to be read by fontROM. We layed out all 4800 tiles in row major order inside of a inferred BRAM through having it read them from a .txt file where we mapped out the background. Our background graphics were strongly influenced by the original Pac-Man maze as we used the image below from GameInternals "Understanding Pac-Man Ghost Behavior" as inspiration to design each tile needed.



Pac-Man

The next step to achieve functionality was to get Pac-Man properly traversing the maze smoothly and true to his original movement style. To do this we took user input from the keyboard using keys 'W' (up), 'A'(left), 'S' (down), and 'D'(right) to control his movement. We then used one-hot encoding in an array to mark a high bit to see the last pressed key code. This

was a 4-bit array. Bit[3] represented ‘W’, bit[2] represented ‘S’, bit[1] represented ‘A’, and bit[0] represented ‘D’. The one hot encoding allowed Pac-Man to traverse the last pressed direction when able to, this means he will continue in his current direction if the last pressed direction is not possible at that time. If Pac-Man does not have a valid direction pressed and collides with a wall he will then stop. We were able to track Pac-Man's movement and wall collision detection by coding in all the possible bounds of his movement along with his possible movement directions. We were able to locate and base our bound checks off of the map we created below.



Ghost’s

Our Pac-Man game had three ghost’s, Clyde, Stinky, and Tinky. Each ghost had the same bound checks as Pac-Man and used the same one hot encoding method, however they had some fundamental differences. First, the ghosts movement was determined based off of Pac-Man’s location rather than keyboard input. Clyde’s movement was determined by Pac-Man’s center location. If its X coordinate is further away it will try to move closer in that direction, otherwise,

if its Y Coordinate is further away from Pac-Man it will move in that direction. Tinky's movement direction was determined by Pac-Man's X coordinate. If the X-coordinates were equal, then it would go off of his Y direction. Lastly, Stinky's movement direction was the opposite of Tinky's as it was determined by Pac-Man's Y coordinate. If Stinky's and Pac-Man's Y-coordinates were equal, he would then go off of Pac-Man's X coordinate. In contrast, if it is in a time period where Pac-Man can eat the ghosts, they will move away to further themselves in these directions. Additionally, the main difference in Pac-Man's and the Ghost movement is that Ghosts could only switch directions at intersection locations(locations it can turn a different direction). Along with that, to prevent the ghosts from ever stopping, if their next move at an intersection is invalid (create wall collision), they each have a unique default move for each intersection to allow for continual traversal. Lastly, when Ghosts are set in the Ghost Pen, whether at the start of the game or after being eaten, they are allowed to leave when a power Pellet is not active and Pac-Man is not at his spawn point.

Score Tracking Logic / Pellet Eating

To keep track and generate Pac-Man's score we used a module to calculate the current tile Pac-Man's center was located at in our maze. We calculated the tile through the row major order equation (PacY%8)*80 + PacX%8. The module would then read this tile from the BRAM and if it corresponded to a tile with a pellet in it, a score signal would be asserted. This would then modify the tile in BRAM by converting it to an 8x8 black tile and the score would be updated in the score register. For every pellet consumed the score increased by two. The output score from the register was then an input to a combinational module that would convert the score from hexadecimal into decimal by breaking up each digit into an individual array to calculate the ones, tens and hundreds place as the max score is 3 digits. These were the equations that were used:

uno = scorein %10 //calculates the ones place

diez = (scorein/10)%10 //calculates the tens place

cien = scorein/100 //calculator the hundreds place

This would then be passed into a color mapper and displayed to the screen.

Collisions

Collisions were determined using a collision detection module. This module would look at the tile that the center of the ghosts and Pac-Man are located in by using the row major equation (Ycord%8)*80 + (Xcord%8). Ycord and Xcord were substituted with the X and Y coordinates of Pac-Man and the individual ghost to calculate each of their own tiles. If any ghost tile was equal to Pac-Man's then appropriate signals would be exerted to signify collision. During a regular scenario where Pac-Man could not eat the ghost the exerted signal would move Pac-Man back into the start location and decrease his life count. If eat_time was high (the signal that was exerted when Pac-Man consumed a power pellet) Pac-Man could collide with the ghosts with no repercussions. The ghosts would then be moved to their starting location and no lives would be lost. Each ghost had a unique one hot encoding in our output signal to determine who had collided.

Speed

Pac-Man and the ghost's speed were determined by the frame clock. Every clock cycle Pac-Man's direction was able to change and he would move in that direction as long as it did not break any maze boundary rules. To make the game more playable the ghosts were given a slower speed. To do this a small counter was used that only allowed the ghosts to move on count one and two, but not on count three, making the ghosts have two-thirds of Pac-Man's speed.

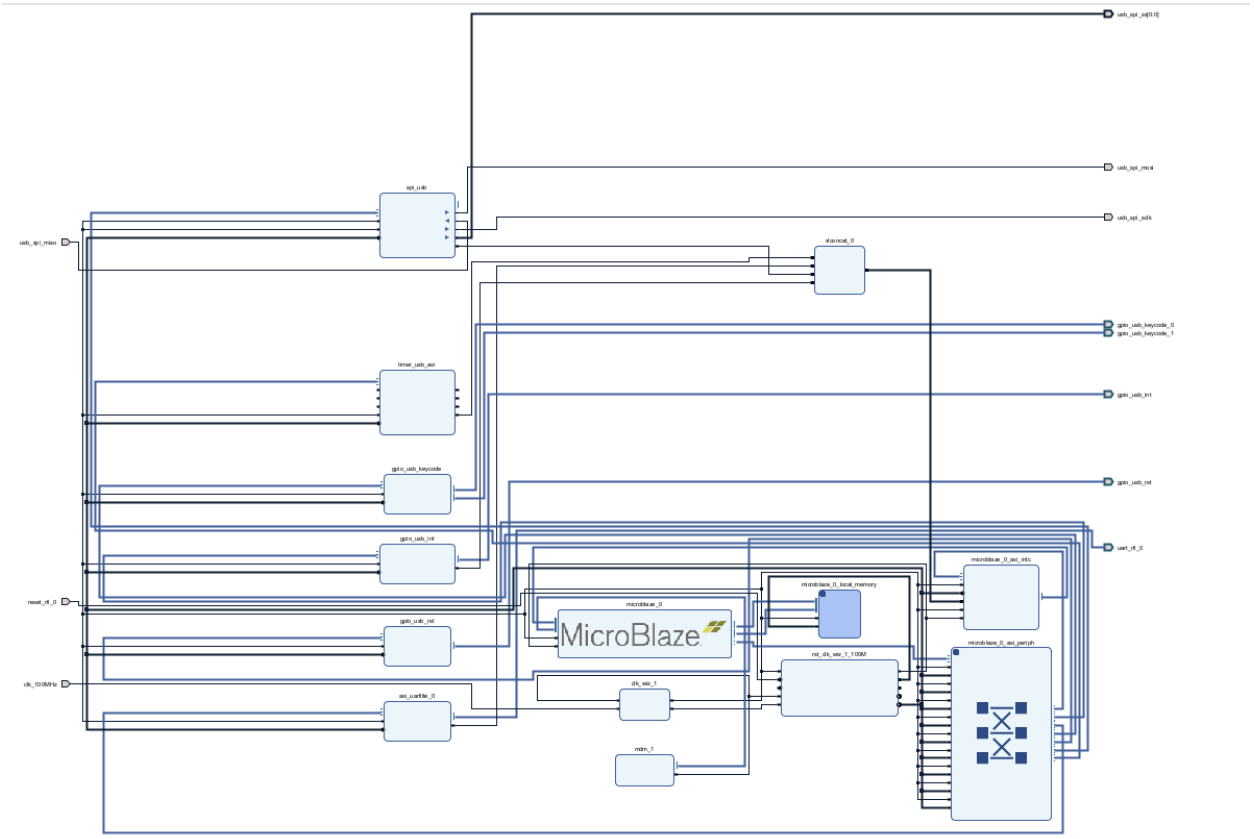
Pac-Man & Ghost Sprites

To display the Ghost Sprites and Pac-Man sprites we used PPU Raster logic. Pac-Man was a 13x13 sprite while the Ghosts were each 14x14 sprites. Both Pac-Man and the Ghosts sprites were designed and made by us. We did gain inspiration for the ghost sprite however from Fred27 on Pragrammable “Getting started on the Verilog”. We made 9 Pac-Man iterations and 2 Ghost sprites that would get animated through. These iterations were all stored in an individual fontrom for each character. Because only Pac-Man or the Ghost’s center are being passed in, we are able to determine if the current pixel being drawn is a part of PacMan or the ghost by offsetting the center and checking if it is within the 13x13 or 14x14 square bound through the equation (drawX >= (PacX -6)) && (drawX <= (PacX + 6)) && (drawY >= (PacY -6)) && (drawY <= (PacY + 6)) for PacMan, and the equation (drawX >= (GhostX -7)) && (drawX <= (GhostX + 6)) && (drawY >= (GhostY -6)) && (drawY <= (GhostY + 7)) for the Ghosts. If it is within these bounds a signal will be asserted with display priority to Tinky, Stinky, Clyde, Pac-Man, then background. We would read our data info from the address line of their respective fontrom through the equation ghost_addr = ((ghostcode *14) + (drawY - (GhostY-6))%14)%32 for ghosts, and the equation pac_addr = ((paccode *13) + (drawY - (PacY-6))%13)%128 for Pac-Man. Paccode and ghostcode are their current animation states. We were then able to determine which pixel was being drawn from each font rom by offsetting the current center of the character through the equation pac_data[12 - ((((drawX - (PacX-6))))%13)] for Pac-Man, and the equations ghost_data[26 - (((2*(drawX - (GhostX-7))) +1)%27)] and ghost_data[27 - (((2*(drawX - (GhostX-7))))%27)] for the Ghosts.The ghosts had two equations due to their multicolor sprites. This meant their fontrom was double in length as it had to store multiple bits per pixel to represent the color to be drawn. These being 00 for the background color, 01 for the eye color, 10 for the iris color, and 11 for its body color. These colors were set to change based

on the ghost, and whether Pac-Man could eat them or not. These equations were all based off of the center being at pixel (8,7) for the Ghosts and (7,7) for Pac-Man. Additionally the fontrom was all stored in little endian order meaning the subtraction from the number of bits in the equation was needed for proper reading.

Written Description and Diagrams of Microblaze System

The following image is of the constructed block design including the Microblaze system. For our final project we used the same construction seen in lab 6.2.



Below is a description of all IP Blocks along with all of the components of Microblaze. Since this is the same as used in lab 6.2 this has been copied from our lab 6 report.

Clocking Wizard

This block has an input of the clock and distributes the signal to all other clock dependent components to make sure they are all synchronized. It has a 100MHz input and outputs two clock signals to be used by the hdmi connection, 25MHz and 125MHz.

Microblaze

The Microblaze is a soft microprocessor IP block that provides the utility of a RISC microprocessor with a modified harvard construction. It has multiple data buses and one unified memory. This microprocessor was configured to microcontroller settings, so we do not have access to any cache.

Microblaze Local Memory

This is used as a memory facility for the Microblaze soft processor. It is a larger bank than what would be on a chip.

AXI Concat

This unit takes the interrupt signals from the timer, uartlite, spi usb, and the gpio usb int and concatenates all the interrupt signals to be fed into the interrupts controller.

Microblaze Interrupt Controller

This unit takes in multiple interrupt signals provided by AXI Concat and transmits these signals to the Microblaze processor.

AXI Timer

Called the timer_usb_axi inside of our block, this peripheral is used as a counter and timer generating interrupt signals. It takes the inputs of a clock and a reset signal.

AXI Periph

MicroBlaze AXI Peripherals provide a standardized interface for the Microblaze soft processor to communicate with other IP blocks within FPGA.

AXI Uartlite

Called the axi_uartlite_0 in our block, this peripheral connects the processor to data inputs. It facilitates asynchronous data transmission between the FPGA and external devices.

AXI Interconnect

This unit connects multiple AXI masters to multiple AXI Slaves. In the case of this lab we have multiple slaves (USB keyboard / HDMI monitor) to connect to the single master (Microblaze).

Modules

mb_usb_hdmi_top

input logic Clk,

input logic reset_rtl_0,

//USB signals

input logic [0:0] gpio_usb_int_tri_i,

output logic gpio_usb_rst_tri_o,

input logic usb_spi_miso,

output logic usb_spi_mosi,

output logic usb_spi_sclk,

output logic usb_spi_ss,

//UART

input logic uart_rtl_0_rxd,

output logic uart_rtl_0_txd,

//HDMI

output logic hdmi_tmds_clk_n,

output logic hdmi_tmds_clk_p,

output logic [2:0]hdmi_tmds_data_n,

output logic [2:0]hdmi_tmds_data_p,

//HEX displays

output logic [7:0] hex_segA,

output logic [3:0] hex_gridA,

output logic [7:0] hex_segB,

output logic [3:0] hex_gridB

This module holds various module instantiations including USB, UART, HDMI, HEX displays, clocking, VGA sync signal generation, VGA to HDMI conversion, color mapping, and all game logic. This module connects all of the following modules and allows for their signals to interact.

Background_ram

input logic [12:0] r_addra, r_addra2,

input logic [12:0] w_addra, w_addra2,

input logic Clk,

input logic wea, wea2,

input logic [8:0] dina, dina2,

output logic [8:0] douta, douta2

This module defines an inferred simple dual-port RAM with two read and write ports. It uses a block memory of 4800 12-bit words initialized from a text file named "backgroundbc.txt". This file contains maze and color info relevant to the background of our game. On each clock cycle, it reads from or writes to the memory based on the input addresses and control signals. It updates the output data based on the read addresses.

pacman

input logic won, lost,

input logic Reset,

input logic frame_clk,

input logic [3:0] collide,

input logic [7:0] keycode,

output logic [9:0] BallX,

output logic [9:0] BallY,

output logic [9:0] BallS ,

output logic [9:0] Ball_X_Motion_next_out,

output logic [9:0] Ball_Y_Motion_next_out

This module controls the movement of pacman. Pac-Man's location is output from this module and the GPIO keycode along with collision detection are inputs. The module was designed by hard coding every possible motion of Pac-Man within the maze to ensure he does not leave the bounds. On reset, when a collision is detected, or when lost or win is high Pac-Man is reset to the starting point of the maze.

Clyde, Tinky, Stinky

input logic won, lost,

input logic Reset,

input logic frame_clk,

input logic [3:0] collide,

input logic [9:0] pacX,

input logic [9:0] pacY,

input logic eat_time,

output logic [9:0] BallX,

output logic [9:0] BallY,

output logic [9:0] BallS

Modules Clyde, Tinky, and Stinky are grouped together into one module description because their functionality is nearly the same. The ghosts hunt Pac-Man down. Clyde follows the minimum of Pac-Man's drawX and drawY. Stinky follows his drawY and Tinky follows his drawX. The clock used to update positions is a 25 MHz frame clock. Additional logic is added to prevent ghosts from getting stuck on corners.

Lives

input logic clk,

input logic reset,

input logic [3:0] collide,

output logic [3:0] livesout,

output logic lost

This module defines a register called lives. The register is initialized to a value of three and each time the collide signal 4'b0001 (meaning a ghost collided with Pac-Man) livesout is decremented. When livesout equal zero the signal lost is set high, ending functionality in all other modules.

scorereg

input logic clk,

input logic reset,

input logic scoreUpdate,

output logic [11:0] scoreout,

output logic won

This module defines a register called scorereg. This register is initialized to a value of zero and every time scoreUpdate is high the scoreout signal is incremented by two points. ScoreUpdate comes from the pelleteparty module defined below. When the score is equal to 600, all pellets are consumed and won is set high, ending the game.

pelleteparty

input logic [8:0] read,

output logic scoreUpdate,

output logic wea,

output logic start_the_count,

output logic [8:0] douta

This module is used to update control signals that define if Pac-Man is consuming a regular pellet (increment score) or if Pac-Man consumes a power pellet (able to eat ghosts). It takes an input read which is a tile read from our BRAM (computed using Pac-Man's coordinates). If the read value is 9'h063 a regular palette is consumed, so write enable is set high, douta is set to 9'h000 to write a blank square in its place, and scoreUpdate is set high. If the read value is 9'h062 a power pellet has been consumed. The same process of writing a blank square into BRAM is done and start_the_count is set high. This starts a counter/fsm that allows Pac-Man to consume the ghosts for 5 seconds. This FSM will be discussed next.

eat_counter

input logic clk,

input logic reset,

input logic en,

output logic eat_time,

output logic white

This module implements an eat counter with a state machine. It counts time elapsed while a condition the enable signal "en" is active. This signal comes from start_the_count located in pelleteparty. It transitions through states IDLE, RUNNING, and DONE. The counter variable

tracks elapsed time for five seconds and resets if enable goes high again. During the last 1.5 seconds white is flipped on and off every 0.25 seconds to signify the timer is almost over. The eat_time signal indicates whether pacman can eat this ghost, which changes the interactions in the collision module. When the counter reaches a threshold, it transitions to the DONE state. For more information view the state diagram and counter section of this report.

collision

input logic[12:0] pac,

input logic[12:0] clyde,

input logic[12:0] stinky,

input logic[12:0] tinky,

input logic eat_time,

output logic[3:0] collide

This module is used to detect collisions between the ghosts and Pac-Man. Collisions are determined using the 13 bit tile address pac, clyde, stinky, and tinky, along with the eat_time signal showing that Pac-Man ate a power pellet and can now eat the ghosts. The output collide is a 4 bit one hot encoding. If eat time is high and Pac-Man is on the same time as the ghost collide signals this and the respective ghost gets moved to the middle of the map. If eat time is low and a ghost collides with Pac-Man he moves to the starting location and loses a life.

Scoretodec

input logic [11:0] scorein,

output logic [3:0] cien,

output logic [3:0] diez,

output logic [3:0] uno

This module is used to convert a hexadecimal score to a decimal value to be mapped to the screen by color mapper. The input score comes from the score register. Each output array is used to mark the specific value zero through nine of each decimal place in the score. For instance if score in was equal to 0x113 the module would convert this to decimal (275) then palace the respective value in the correct array. Cien would be 0x2, Diez would be 0x7 and uno would bet 0x5.

vga_controller

input logic pixel_clk, // 50 MHz clock

input logic reset, // reset signal

output logic hs,

output logic vs,

output logic active_nblank, // High = active, low = blanking interval

output logic sync, // Composite Sync signal. Active low. We don't use it in this lab,

output logic [9:0] drawX, // horizontal coordinate

output logic [9:0] drawY ; // vertical coordinate

The vga_controller module is designed to generate VGA signals for a standard 640x480 resolution display. It provides horizontal sync (hs), vertical sync (vs), active_nblank, and drawX/drawY coordinates for the display. The module counts horizontal and vertical pixels and lines, generates sync pulses, and controls display enable based on the specified resolution parameters. Additionally, it disables the composite sync signal and ensures that only the pixels

within the defined display area are activated. This module was not modified for this project. The code was identical to lab 6 and lab 7.

ghost_fsm

input logic clk,

input logic reset,

output logic [1:0] ghost_code

This module is a finite state machine that is used to animate the ghosts movements. An internal counter is used to alternate the ghosts between two animation states, FIRST and SECOND, every 0.25 seconds. The FIRST state outputs a code to color mapper which will have the ghosts legs on the left and the SECOND state will output a code that has the ghosts legs on the right. This fsm starts when the program begins and never ends.

pacman_fsm

input logic clk,

input logic reset,

input logic [9:0] motion_x,

input logic [9:0] motion_y,

input logic won,

output logic [3:0] pac_code

This module is responsible for controlling the animation state of a Pac-Man character . It takes inputs such as motion in the x and y directions, along with a flag indicating whether the game has been won. Based on these inputs and the internal counters, the module determines the

appropriate animation state for Pac-Man, for various directions of movement and open/close mouth animations. The output pac_code represents the current animation state and can be used to alternate images in a color mapper.

color_mapper

input logic [9:0] drawX,

input logic [3:0] paccode,

input logic lost, won, eat_time, white,

input logic [9:0] drawY,

input logic [9:0] BallX, BallY, Ball_size, Tinky_size, TinkyY, TinkyX, StinkyX, StinkyY, Stinky_size,

input logic [9:0] ClydeX, ClydeY, Clyde_size,

input logic[8:0] code,

input logic [3:0] cien, diez, uno,

input logic [3:0] livesout,

input logic [1:0] ghostcode,

output logic [3:0] red, green, blue

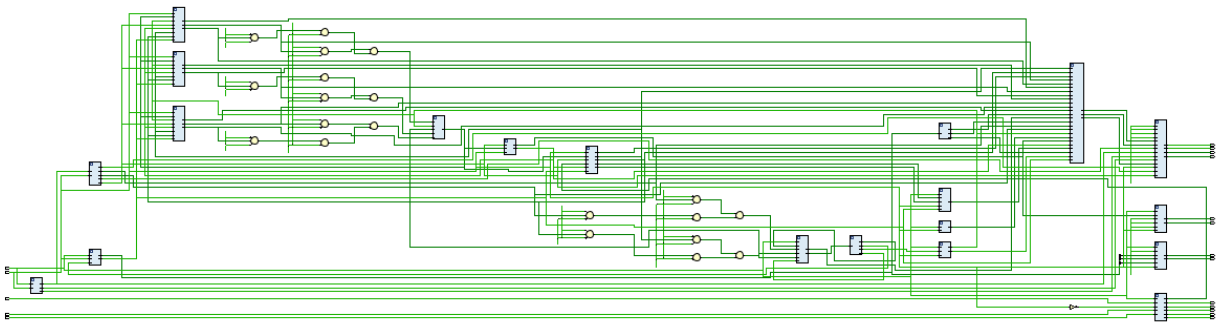
The module colors the background, ghosts, score, and Pac-Man. This module determines the color of a pixel based on the position of the sprites and the draw area. It accesses font readonly memory to select an appropriate sprite from here it is read and displayed to the screen. Additional internal logic is added to address transparency and overlap between the ghosts. The priority order of what is displayed is Tinky, Stinky, Clyde, Pac-Man, then the maze background.

Software

In this lab the same software as lab 6.2 was used. We will not go over code since this was done in lap 6 report, but instead a higher level overview. AXreg_wr, MAXreg_rd, MAXbytes_wr, and MAXbytes_rd, all facilitate communication with the MAX3421E chip via SPI protocol. MAXreg_wr writes a single byte to a register by setting the appropriate command byte and utilizing SPI transfer, while MAXreg_rd reads a byte from a register using similar methods. MAXbytes_wr enables bulk write operations by transmitting multiple bytes to a register, and MAXbytes_rd facilitates bulk read operations for retrieving multiple bytes. Each function begins by selecting the SPI instance, constructs necessary data arrays, performs SPI transfers, handles potential errors, and deselects the SPI instance afterward.

RTL Design

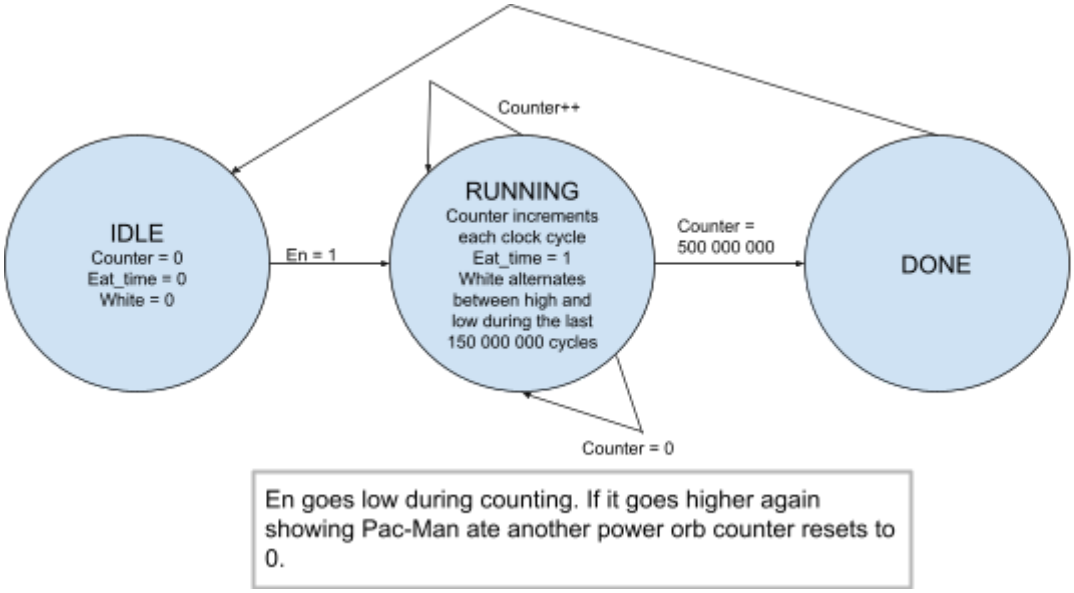
Here is the RTL diagram of the entire Pac-Man game.



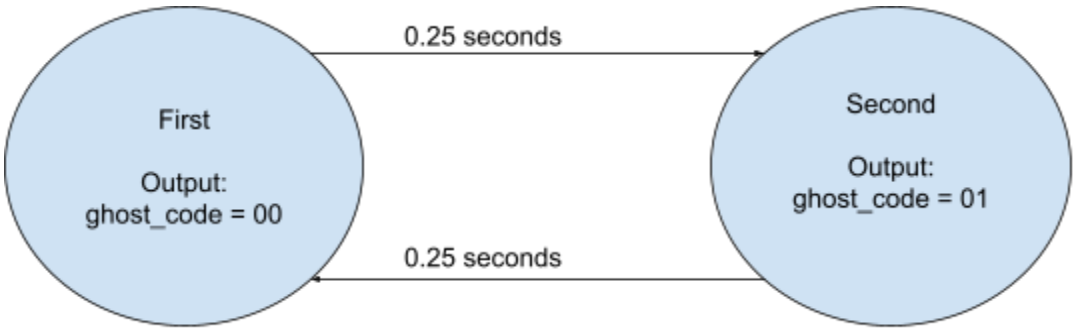
State Diagrams and Counters

The first image in this section is the counter designed in the eat_counter module. The purpose of this is to count for five seconds and during this period a signal eat_time is asserted allowing Pac-Man to collide with ghosts and not lose a life, but instead cause them to move to the middle. When the counter is over 350 000 000 clock cycles, signifying 3.5 seconds has

passed an additional signal white is asserted in 25 000 000 clock periods to allow the ghosts to flicker between blue and white. Below is a drawing of this state machine / counter.

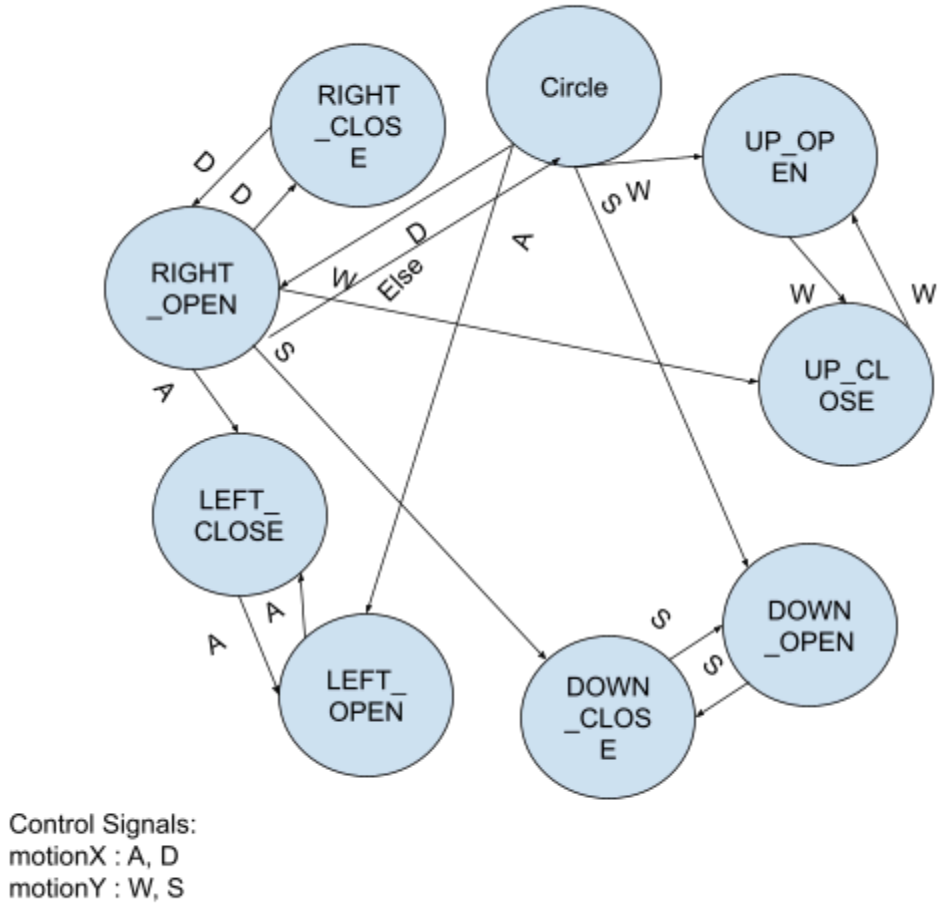


The following image is the ghost state diagram. It has no inputs other than reset which brings it to the first state. Every 25 000 000 clock cycles or 0.25 seconds it switches states to the next animation state. This cycle repeats until reset is high, bringing it back to FIRST.



The following image is the state diagram for Pac-Man's animations. This state machine is controlled by the motionX and motionY signals from the Pac-Man module. Every 0.5 seconds or 50 000 000 clock cycles the animation state updates. If motionY is -1 (w) we move into the UP_OPEN or UP_CLOSE state. Open and close just signify if Pac-Man's mouth is open or closed, it is always the opposite of the previous state. The same applies for all other directions.

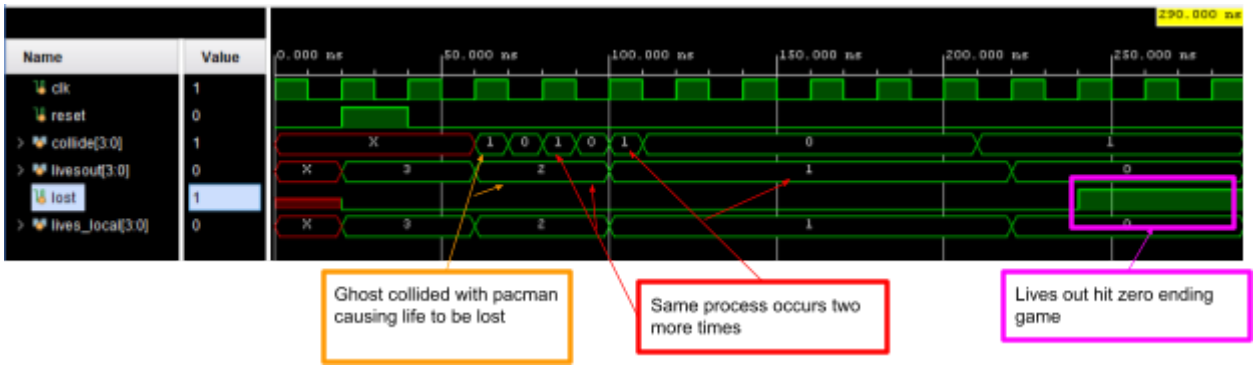
motionY is +1 move to down, motionX is -1 move to left animation, and motionX is +1 move to right animation. If none of these are high go to the circle state. This just turns Pac-Man into a yellow circle for when he has stopped. For the purpose of being more clear on the functionality of the state machine only the RIGHT_OPEN and RIGHT_CLOSE have been fully connected.



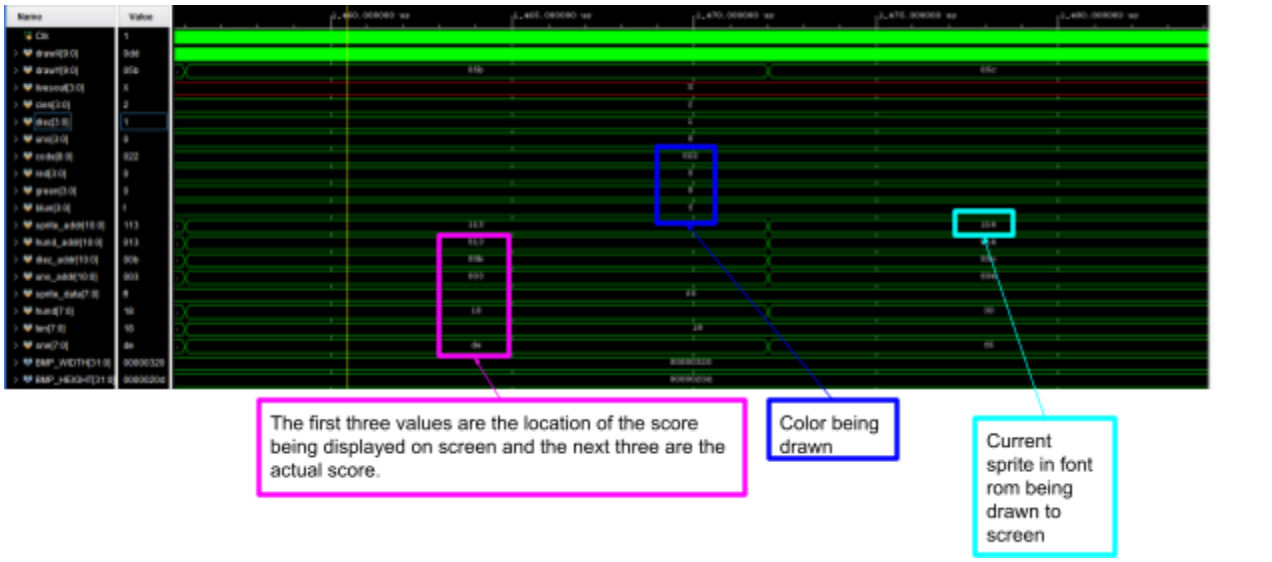
Annotated Simulations

In this section we will be going over some of the test benches and simulations used to design Pac-Man. The first test bench was designed to test the live register. A similarly constructed test bench also tested the score register. This test bench was used to debug the error

where lives would not properly update due to the collision signal being high multiple clock cycles causing all lives to be lost at once. This was fixed by using a slower clock for the register.



The next test bench was used extensively for our color mapper function. This test bench was used to pick up errors in our font rom when designing the maze background..



Design Statistics

LUT	13844
DSP	10
Memory (BRAM)	10.5

Flip-Flop	3077
Frequency	0.14382281029 MHz
Static Power	0.076W
Dynamic Power	0.443W
Total Power	0.519W

Conclusion

In our final project, we designed and implemented a Pac-Man game. The project utilized a Microblaze processor, GPIO peripherals for keyboard input handling, and BRAM for memory storage to accommodate intensive graphical features. The game allowed players to control Pac-Man's movement using a keyboard as he traverses a maze to eat pellets while avoiding ghosts. Power pellets enabled Pac-Man to temporarily turn the tables by allowing him to eat the ghosts. The game ended when either Pac-Man consumed all pellets or lost all lives to the ghosts. The game's design was influenced by the original Pac-Man maze, with 8x8 tiles forming the background and sprites representing characters. Key modules such as pacman_fsm controlled Pac-Man's animation state, while others managed ghost movement, collisions, scoring, and display control. The overall system incorporated various IP blocks, including clocking, Microblaze, UART, and HDMI peripherals, and used a VGA controller for display generation. Simulations and test benches were employed to validate and debug the system's functionality. This project was an interesting capstone to the class and ran to our liking.

Sources of Inspiration for Graphic Display:

1. https://gameinternals.com/understanding-pac-man-ghost-behavior
2. https://community.element14.com/challenges-projects/project14/programmable-logic/b/blog/posts/pragrammaball---getting-started-on-the-verilog