ECE 385

Spring 2025

Final Project

Pac-Man

Jacob Torry, Logan Wonnacott

Lab Section: Friday

XJ

Introduction

For our final project we decided to recreate the classic arcade game Pacman. Pacman is a game where you are a circular yellow character who must navigate a maze and eat all the pellets to progress to the next level. While eating the pellets you are being chased by four ghosts; red, blue, pinky, and orange (Blinky, Inky, Pinky, and Clyde respectively). The goal for each level is to eat all the pellets without being caught and running out of your three lives. If Pacman eats a power pellet the ghosts turn white and run away from Pacman because Pacman is now able to eat the ghosts to add to the score.

We thought this would be a good way to incorporate all of the concepts that made up our previous labs to make a fun and classic game. This project consists of a Microblaze processor that controls the keyboard inputs for movement and the C code for all game logic. The C code controls everything from updating the sprite positions to handling when all the pellets have been eaten. We also used the HDMI IP with BRAM that we designed in Lab 7 to handle the map/maze rendering, sprite rendering, and score rendering.

Description of Design Process (Appendix)

Maze Generation

The first thing that we decided to focus on was generating the map and maze. This included the static text at the top of the screen for the “1UP” and “HIGH SCORE” and the blue maze itself as seen in our reference image in Figure 1. This is the image we used in order to get the correct dimensions, memory and pixel calculations, and detailed wall design.

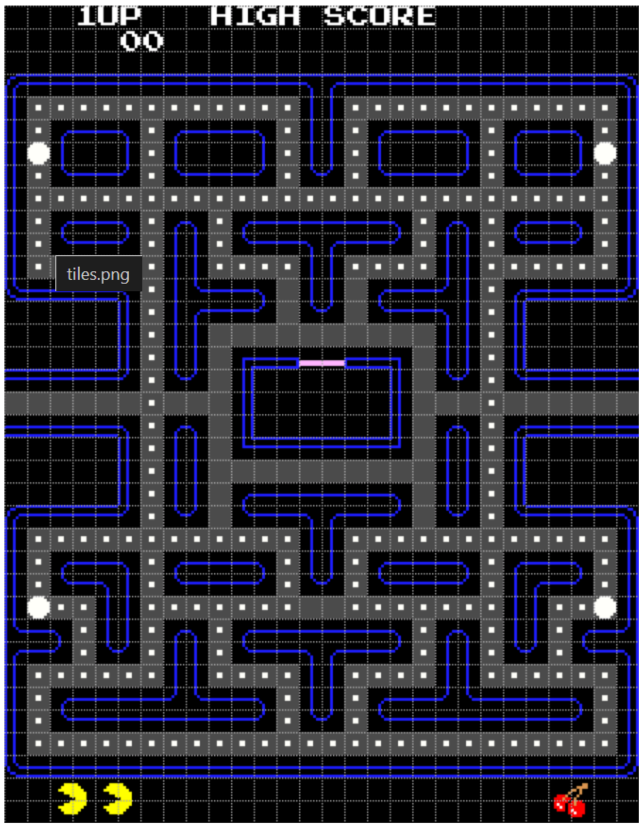


Figure 1. Pacman Maze

The generated screen dimensions are 28 horizontal tiles by 36 vertical tiles, where each tile is made by 8 pixels by 8 pixels resulting in a rendered block of 224 pixels by 288 pixels. The maze itself is 28 tiles by 31 tiles leaving 5 rows of tiles for the lives and score. We also used this image to find the size of the wall, pellets and power pellets in pixels for accurate detailed rendering. To make sure that we designed it to the level of accuracy we wanted we ran a simulation.

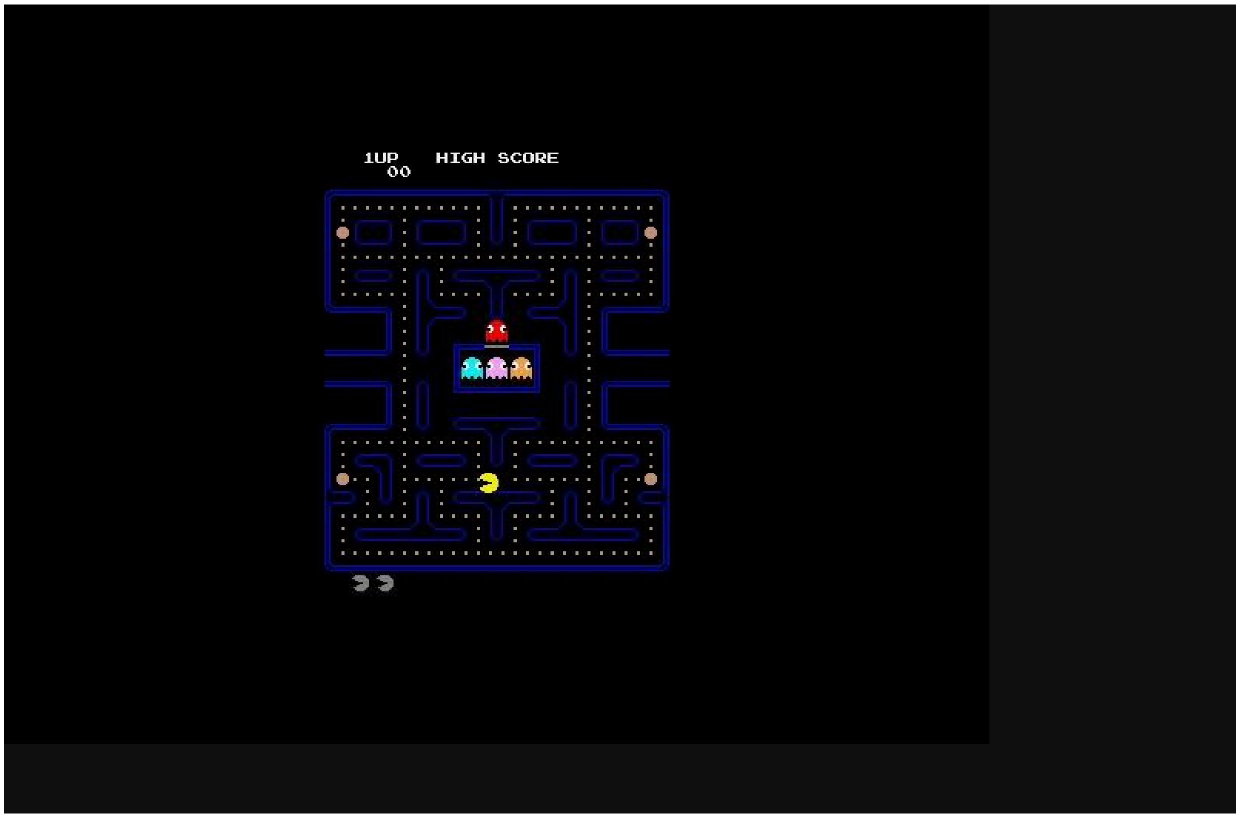


Figure 2. Maze and Sprite Drawing Simulation

This precision was accomplished by creating a bunch of tile codes for each unique tile in the reference image, this way we only need one code to represent all 240 pellets. We then hardcoded our BRAM with these tile codes making sure that the maze was correctly initialized on startup every time (see tilemap.sv). The tilemap just tells the FPGA which tile goes where but in order to get the detail, we adapted the font rom from Lab 7 to design each tile to pixel by pixel level accuracy. The result was something like this, the BRAM would return a tile code based on an address calculation. That address would then be used to find the corresponding pixel data from the font rom. This is the exact same process that Lab 7 used to get character codes from BRAM and font rom to render the text, the only major difference is that we had to change to 8x8 fonts rather than 16x16.

// Tilecode = x03
8’b0000**1111**
8’b00**11**0000
8’b0**1**000000
8’b0**1**000**111**
8’b**1**000**1**000 ==>
8’b**1**00**1**0000
8’b**1**00**1**0000
8’b**1**00**1**0000

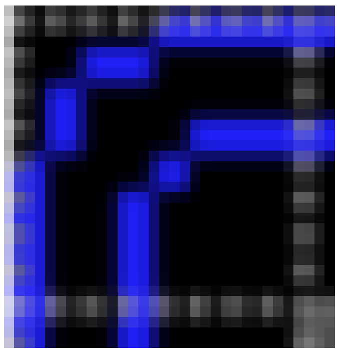
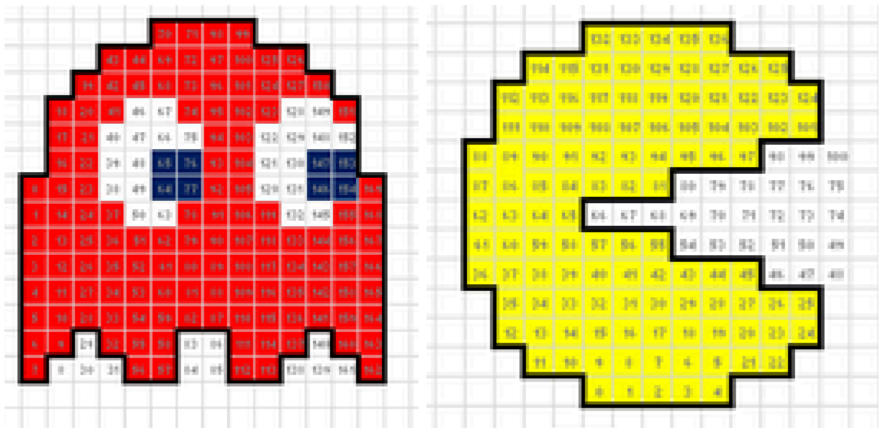


Figure 3: BRAM tile code to font ROM example

This BRAM to font rom process is used for rendering the maze walls, text, score, pellets, and lives. We chose to use BRAM because we needed the ability to write to the tilemap whenever a pellet was eaten or the score needed updating.

Sprite Generation

Pacman and the Ghosts are sprites, implemented using ROM, that are rendered over the tilemap. The Pacman sprite is 13 pixels by 13 pixels but has four different versions; one for each direction he is traveling (up, down, left, right). We store all versions in one ROM by assigning each direction a number and use that to offset and generate the correct orientation of pacman. The ghosts, however, are 14 by 14 pixel sprites with three colors so we needed a separate ROM for them. Unlike Pacman whose color can be represented with 1 bit per pixel for color encoding; either 1 (yellow) or 0 (transparent). The ghosts have 3 colors so we needed a minimum of 2 bits per pixel to represent colors.



Sprites are rendered when the current drawX and drawY fall within the bounds of each sprite. We determine the bounds for each sprite by reading from movement registers that are updated from our game logic. The movement registers are repurposed palette registers from Lab 7 that hold the X,Y pixel coordinates for each sprite. Our system works by updating X,Y in the C code and AXI writing the new values to the respective movement registers which the render uses to decide when to render the ghost, pacman, or the maze in that order (priority ordering).

Game Logic

Now that we have discussed how we thought about rendering the game and characters (through system verilog), we need to discuss the logic implementation. We decided that the easiest way to implement the game logic was to use C code because we could then use Lab 6 as a template so that we already have the USB SPI protocol to access the keyboard inputs. This is where most of our time spent designing went. We often worked on getting one feature working at a time, eg. pacmans movement -> pellets eating and score -> map reset -> ghosts.

Helloworld.c

In hellworld.c is where the program launches from. When the program starts, the default Lab 6 functions run, initializing the MAX3421E and USB. Once a keyboard is detected the main function calls the game loop. In our game loop we first initialize all necessary variables like score, lives, etc. We also set the starting positions for Pacman and the ghosts as seen in Figure 2. After the initializations we jump into the main game loop where we are constantly polling the keyboard for inputs from WASD so we can determine Pacmans movement from the user.

Pacmans movement: When W, A, S, or D are pressed we update Pacmans X and Y position and use AXI write to write his new location to the movement registers. We also write his direction (left, right, up, down) to a movement register so that we can render him in the correct orientation. We also include additional logic so that Pacman is centered in the paths and does not render over the maze walls. We also have a function that checks the tile in the direction that pacman is moving to determine if that is a valid move for Pacman to make.

Ghost movement: Handled in ghost.c but is called in the main game loop function.

Pellet Handling: We call an AXI read to read the current tile that pacman is on and check its tile code. If the code is a pellet we AXI write a blank/path tile to that location and increment the score. If the pellet is the power pellet we set a scared flag in the ghost struct and also AXI write to a movement register so that our render can change the ghosts color to white indicating scared mode. We also define a condition where we check the current pellet count to see if all the pellets have been eaten. If so, we reset the pellets in the maze by rewriting the pellet tile to BRAM (see tilemap.c), and reset Pacman and the ghost positions. This is the equivalent of increasing to the next level.

Collision Handling: Another important component of the game is when Pacman and a ghost collide. If the ghost is scared and they collide then we call ghost_die() which sends the ghost back to the starting box and increments the score by 200. If the ghost is not scared and they collide then we decrement the lives by AXI writing blank tiles to the life area and reset Pacman and the ghosts positions. If Pacman dies three times we reset the life count, pellet counts, clear the score (not the high score), and reset the sprite positions for a new game.

Score Updating: Keeping track of the score was pretty straightforward. Essentially we assign global variables called score and highscore. Anytime that score is greater than highscore we update highscore to match. Anytime we eat a pellet, powerpellet, or ghost we update the current score. Anytime Pac-Man ran out of lives or the game ended we would reset the score.

Writing the score to the screen was a bit trickier, however. We made a couple helper functions. The first was called draw_number. It took a tile coordinate and a number and printed that number to that specific tile. But this solution is only good for one digit. So we needed to extend this. To do this we added two additional functions called update_score and update_high_score. What these do is they both take in the score and high score respectively and we do some arithmetic to isolate each digit. Once we have these digits we draw all 5 of the possible digits at the same time using draw_number.

Each of these mechanics utilize common functions like read_tile which performs an AXI read from BRAM or write_tile/number which is an AXI write to BRAM.

Ghosts.c

In ghost.c we handle all of the ghost logic. The first thing we would like to mention is our set up of the ghost struct. Each ghost has a set of its own unique variables. These being x, y, direction, target_x, target_y, scared, move, state, boc_time, release_delay, relese_path, path_index and path_timer. The decision to use a struct ws critical because then we could easily create four ghost objects rather than needing separate functions for each ghost.

The first important function I would like to mention is the initialization function we call init_ghost. We call this function at the very beginning of the game as well as whenever we complete a level or if the ghost is consumed by pacman. This function essentially sets all the member variables of the ghost structure.

The core function is the update_ghost function, which is in charge of determining the next position of the ghost and setting the X,Y struct fields to that new opposition so we can AXI write the new location to the movement registers for rendering. First we check the state of the ghost. If the ghost is in the IN_BOX state it does not move until the box_timer reaches its desired value. After that, depending on the ghost, it follows a hardcoded path to navigate outside of the box and snap to the maze paths (ALIGN_PATH state). Once the ghost exits the box its state is set to ACTIVE and determines which chasing algorithm it will use based on the color of the ghost and sets the targetx and targety struct in the ghost.

- Red (Blinky) chases pacman directly by targeting the current tile that pacman is in. When Blinky is scared he scatters to the top left of the maze.
- Pink (Pinky) aims for four tiles in front of pacman to try and cut pacman off. When Pinky is scared she scatters to the top right of the map.
- Blue (Inky) in the original games uses vector math to take red's position and aim for two tiles ahead of pacman from red's location. However, our implementation was causing Inky to move very glitchy so we changed the algorithm. Instead in our implementation Inky tries to mirror pacmans movements across the mazes center. When Inky is scared he scatters to the bottom left of the map.
- Orange (Clyde) directly chases pacman when he is far away, but if clyde gets within a certain distance of pac man he runs away. When Clyde is scared he scatters to the bottom right of the map.

After setting the target location based on their respective algorithms we use a breadth first search (BFS) algorithm to find the shortest path from the ghost to its target destination. This was an extremely important addition because without it the ghost tried to calculate distance through walls and would get stuck. When the BFS returns with the next direction, the ghost “steps” in that direction resulting in smooth and accurate ghost movement to the original games.

There are a few additional functions that help keep the ghost centered in the path or their sprites dont clip into the walls and also checking to see if the tile we want to target is a valid tile that they can move to. The ghost file always utilizes functions like read_tile so that we can check valid moves.

Tilemap.c

This file is essentially a 28*36 length 1D array. This array called tilemap_reset holds the tiles for the background of the game. We use this anytime we reset the map.

Deliverables from Proposal

We believe that we delivered the baseline of our expected difficulty of 5/10. We think we replicated the original game very well. Unfortunately we were unable to incorporate additional features due to time constraints. However our initial proposal stated we would simply use RNG to incorporate the ghosts movement. But eventually we were able to replicate some of the “AI” as described by the original game. Using BFS we allowed the ghosts to have different personalities. This is why we and the game creators call this “AI”. If we were given more time I think it would have been cool to implement actual AI with Q learning or some other form of true artificial intelligence. It would have also been really cool to implement a joystick for controlling pacman rather than just keyboards inputs to make the game feel more authentic, but early finals and the initial set back created a time constraint that didn't permit that to be an important feature.

Block Diagram

This is the block diagram for our Pacman game. It is basically just a mashup for the Lab 6 and Lab 7 block diagrams.

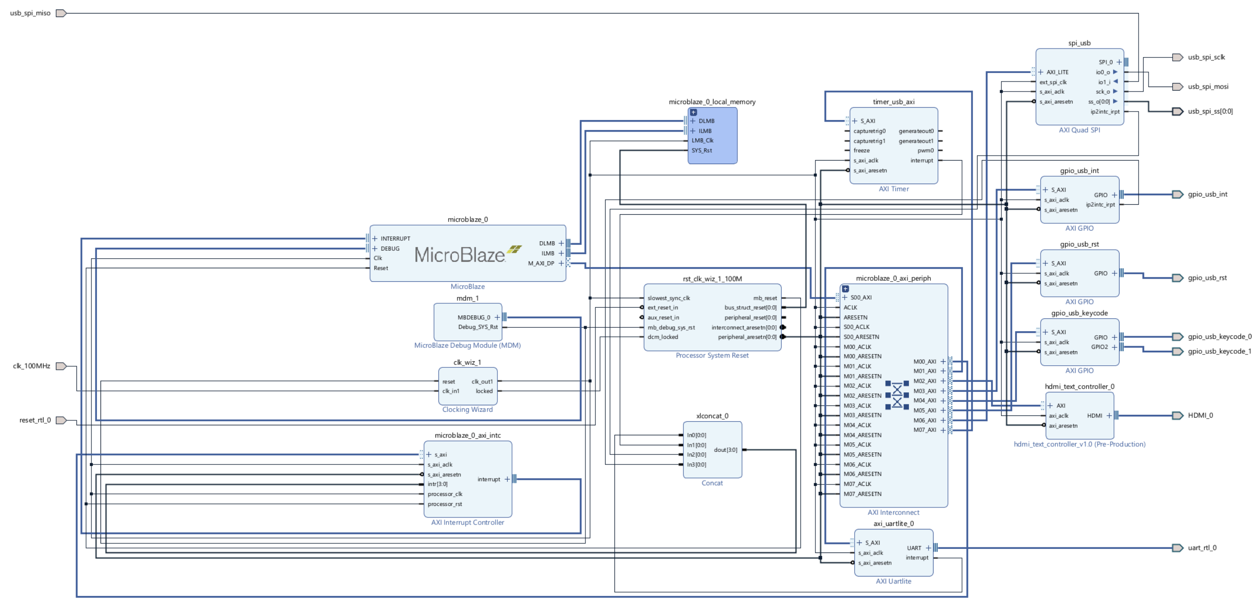


Figure 4.

Module Descriptions

<u>Common SV Modules</u>
Module: font_rom.sv
Inputs: [10:0] addr
Outputs: [7:0] data
Description: This module holds the ROM for the map of PacMan.
Purpose: The purpose of this module is to tell us whether a specific pixel is supposed to be on or off depending on the current position within the map. As a parameter to this module we pass a specific address within this ROM and as an output we get a byte of data which tells us an entire byte of data

Module: ghost_sprite_rom.sv
Inputs: [10:0] addr
Outputs: [1:0] data
Description: This module holds the ROM for the ghosts of PacMan.
Purpose: The purpose of this module is to tell us the color of a specific pixel for whenever we are printing the ghost to the screen.. As a parameter to this module we pass an address within this ROM and as an output we get 2 bits of data which allows us to use a total of 4 colors within the sprite

Module: pacmant_sprite_rom.sv
Inputs: [10:0] addr
Outputs: [1:0] dir
Description: This module holds the ROM for PacMan.
Purpose: The purpose of this module is to tell us the color of a specific pixel for whenever we are printing Pac-Man to the screen. This rom holds an image of Pac-Man looking in the 4 directions. Up, Down, Left and Right. As a parameter to this module we pass an address within this ROM and as an output we get 2 bits (only 1 useful bit) of data which allows us to print Pac-Man.

Module: hdmi_text_controller_v1_0_AXI
Inputs:[9:0] drawX, [9:0] drawY, S_AXI_ACLK, S_AXI_ARESETN, [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_AWADDR, [2 : 0] S_AXI_AWPROT, S_AXI_AWVALID, [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_WDATA, [(C_S_AXI_DATA_WIDTH/8)-1 : 0] S_AXI_WSTRB, S_AXI_WVALID, S_AXI_BREADY, [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_ARADDR, [2 : 0] S_AXI_ARPROT, S_AXI_ARVALID, S_AXI_RREADY
Outputs: [3:0] R, [3:0] G, [3:0] B, S_AXI_AWREADY, S_AXI_WREADY, [1 : 0] S_AXI_BRESP, S_AXI_BVALID, S_AXI_ARREADY, [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_RDATA, [1 : 0] S_AXI_RRESP, S_AXI_RVALID,
Description: This module is our implementation of the AXI handshake. This includes our FSM counter which takes into consideration the delay for accessing BRAM.
Purpose: The purpose of this module is to take drawX and drawY as inputs, access the according BRAM or palette registers and output the RGB of the specific pixel we are trying to

draw. Additionally, here we handle all of our drawing logic. Similar to in lab 7 we were able to calculate if the current pixel was where a ghost or Pac-Man was supposed to be. And if we were within that then we would draw the sprite accordingly.

Module: vga_controller
Inputs: pixel_clk, reset
Outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY
Description: This is the VGA controller
Purpose: The purpose of this module is to synchronize the 480 by 640 VGA display

Module: hdmi_text_controller_v1_0.sv
Inputs: axi_aclk, axi_aresetn, [C_AXI_ADDR_WIDTH-1 : 0] axi_awaddr, [2 : 0] axi_awprot, axi_awvalid, [C_AXI_DATA_WIDTH-1 : 0] axi_wdata, [(C_AXI_DATA_WIDTH/8)-1 : 0] axi_wstrb, axi_wvalid, axi_bready, [C_AXI_ADDR_WIDTH-1 : 0] axi_araddr, [2 : 0] axi_arprot, axi_arvalid, axi_rready
Outputs: hdmi_clk_n, hdmi_clk_p, [2:0] hdmi_tx_n, [2:0] hdmi_tx_p, axi_awready, axi_wready, [1 : 0] axi_bresp, axi_bvalid, axi_arready, [C_AXI_DATA_WIDTH-1 : 0] axi_rdata, [1 : 0] axi_rresp, axi_rvalid,
Description: This module is our top level.
Purpose: The purpose of this module is to connect our AXI driver with our clocking wizard, as well as our VGA controller and in turn out VGA to HDMI converter.

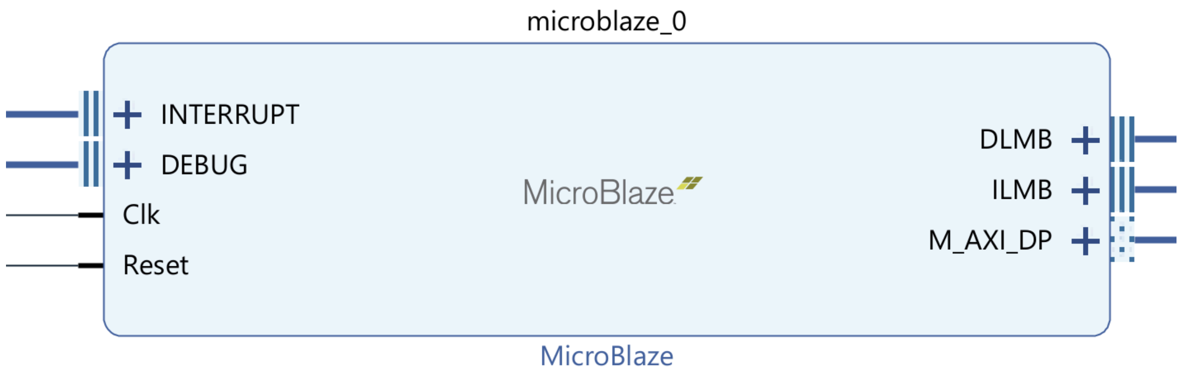
Module: tilemap.sv
Inputs: clk, [9:0] addr_a, [9:0] addr_b, [7:0] data_in_b, we_b
Output: [7:0] data_a, [7:0] data_b
Description: This is a 1100 line long file that is used to directly hardcode and initialize the background of the pacman
Purpose:

Encapsulated Modules

Module: microblaze_0

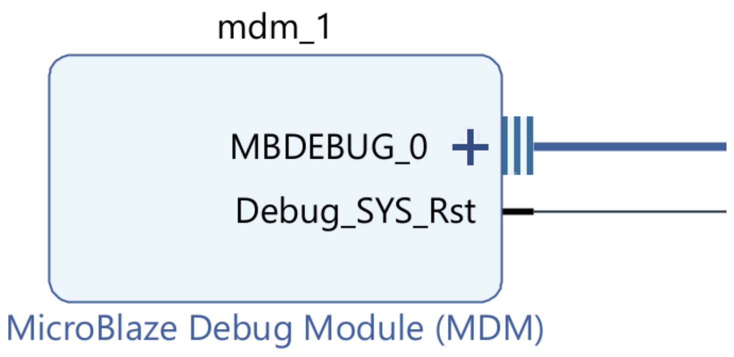
Description: This is the MicroBlaze CPU

Purpose: Basically a small 32-bit processor. The purpose of this module is to allow our system to execute our C code from Vitis.

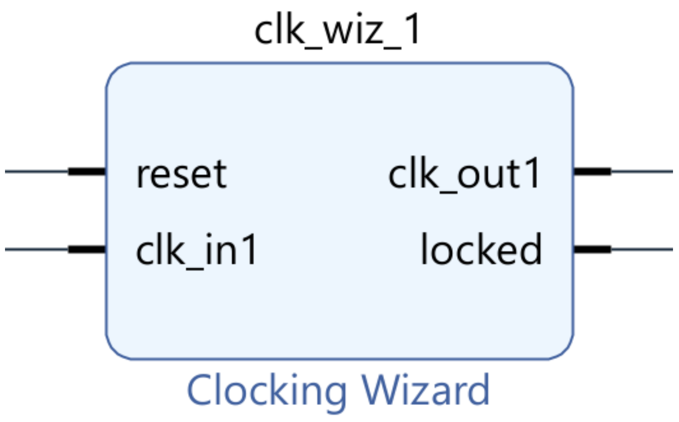


Module: mdm_1

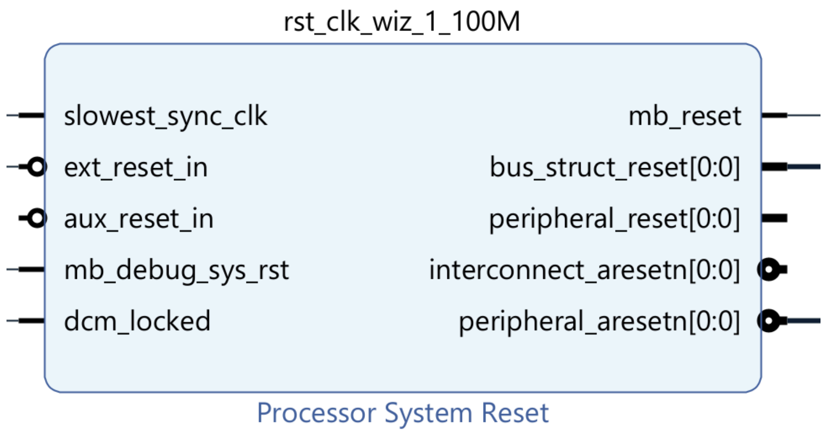
Purpose: The purpose of this module is to allow us to set breakpoints and step through our code in the Vitis debugging tool. Essentially provides a way to make sure the Microblaze is operating correctly, and gives us a way to see the internals if we are debugging.



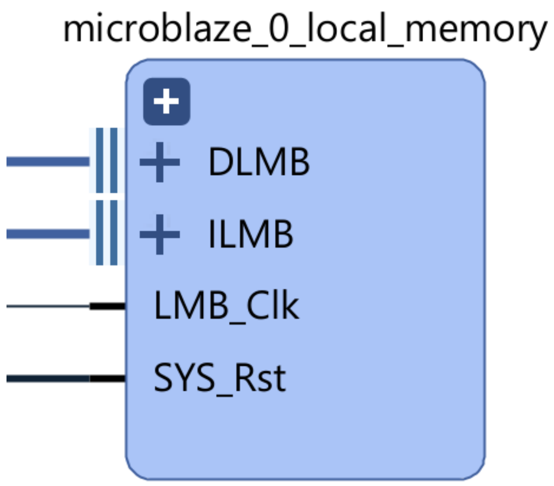
Module: clk_wiz_1
Purpose: The purpose of this module is to generate a common synchronized system clock for the Microblaze and all the other peripherals.



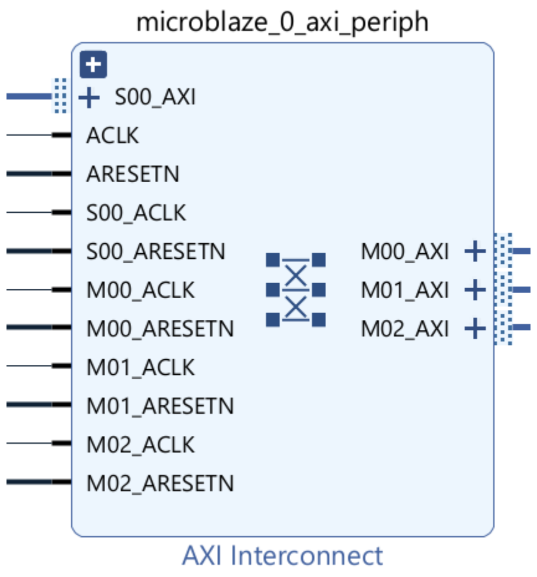
Module: rst_clk_wiz_1_100M
Purpose: The purpose of this module is to manage the reset for all of the IPs and parts of the block diagram and make sure that when reset is pressed that the system is correctly reset and runs clean.



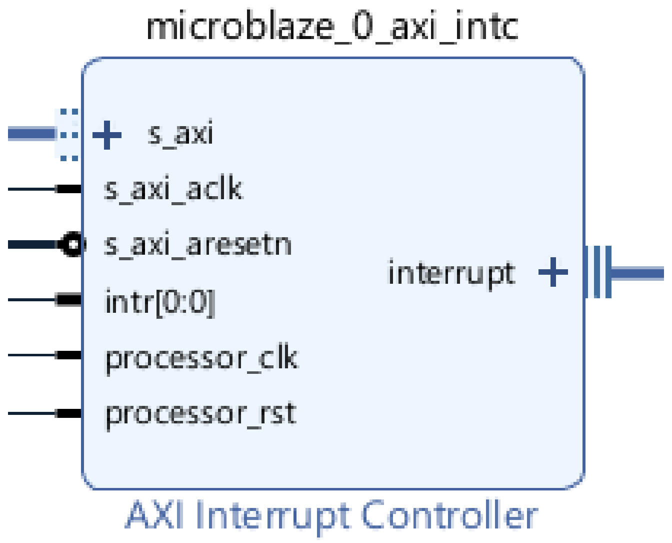
Module: mircoblaze_0_local_memory
Purpose: This is a dual port BRAM memory chip that is used for data (DLMB) and instruction (ILMB) storage. It stores stuff like program code and variables (check post lab question Q7 for what types of segments are in the BRAM). Noet that it is directly accessed by the microblaze and not through AXI.



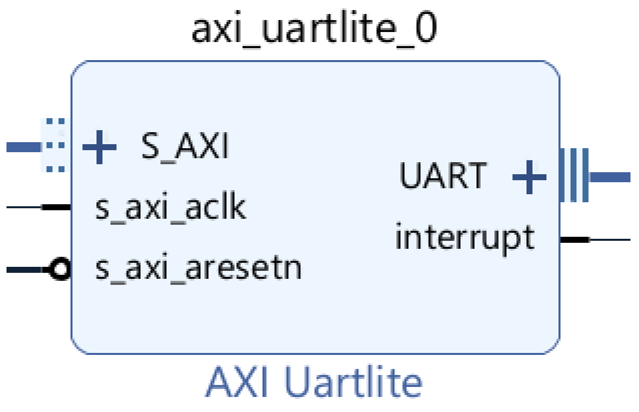
Module: mircoblaze_0_axi_periph
Purpose: The purpose of this module is to connect our MicroBlaze CPU with our GPIO peripherals through the AXI protocol. Its main purpose is to route the AXI signal from the Microblaze (S00_AXI) to the correct peripherals (Mxx_AXI). Almost like a mux.



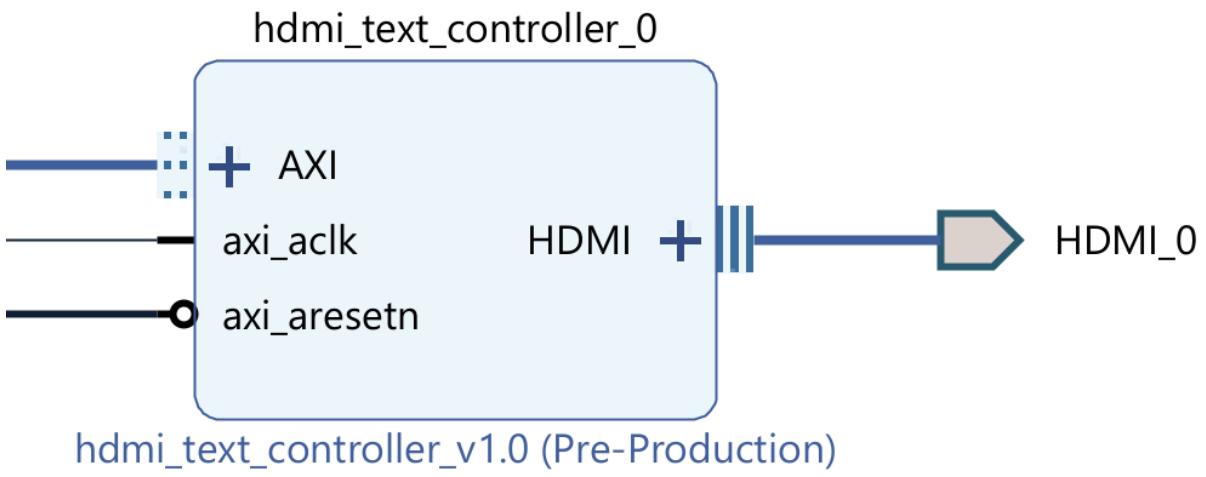
Module: microblaze_0_axi_intc
Purpose: The purpose of this module is to be the AXI interrupt controller. It takes in a bunch of interrupts from different sources (see above) and sends out a single interrupt signal to the system. Keep track of which devices are signaling for an interrupt.



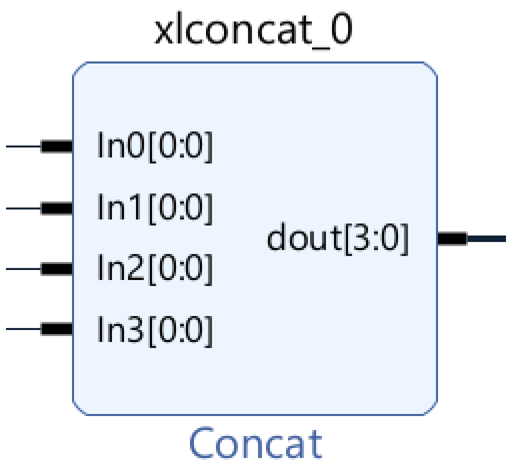
Module: axi_uartlite_0
Purpose: The purpose of this module is to allow our MicroBlaze CPU to communicate asynchronously. This is how our xil_printf functions in the C code are outputted to the serial terminal.



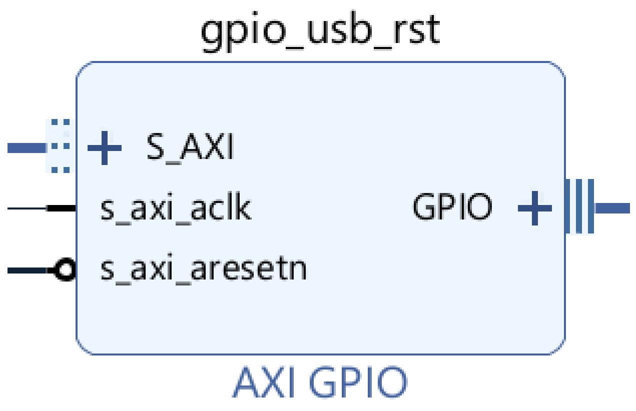
Module: hdmi_text_controller_0
Purpose: The purpose of this module is to take the data that the AXI writes into the internal memory (either registers or VRAM), calculate the RGB values of a specific pixel based on that data, and output a HDMI video signal.



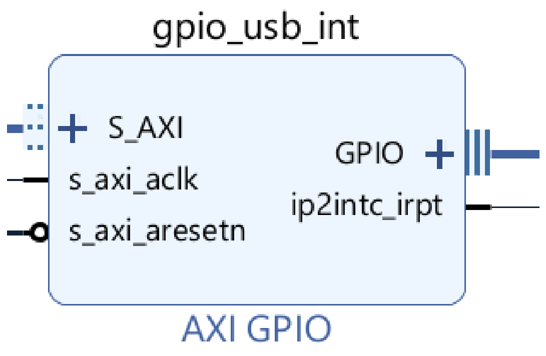
Module: xlconcat_0
Purpose: The purpose of this module is to concatenate 4 interrupt signals from various GPIO peripheral into one signal so keep a clean design.



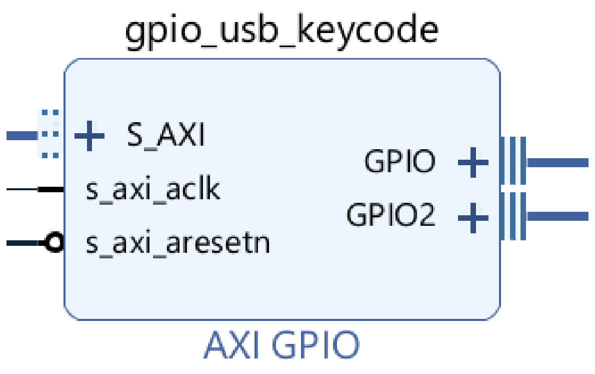
Module: gpio_usb_rst
Purpose: This module sends our reset signal (button 0 on the FPGA) to the USB controller (MAX chip).



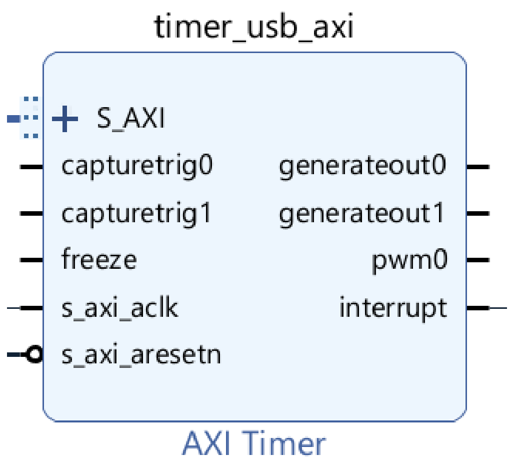
Module: gpio_use_int
Purpose: The purpose of this module is to send an interrupt signal to the Microblaze when the USB device wants to communicate an event. This is what allows us to take a key input that is stored in the keyboards register and send it to the Microblaze.



Module: gpio_usb_keycode
Purpose: The purpose of this module is to read the keycodes from the USB keyboard. By having 2 GPIO ports we can support up to 8 8-bit keycodes, however the keyboard can only support up to 6 simultaneous key presses.

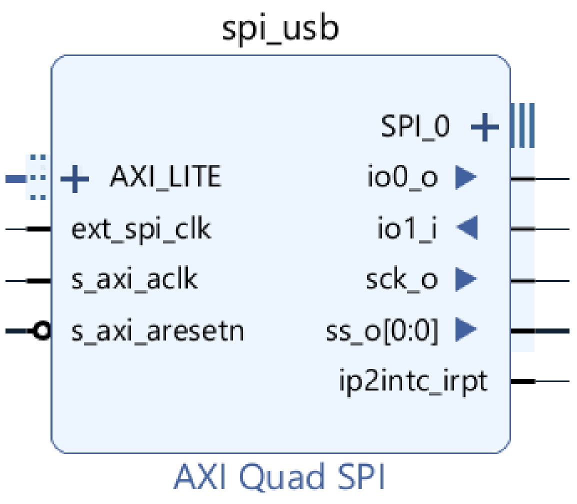


Module: timer_usb_axi
Purpose: The purpose of this module is to generate a timer for the AXI bus. It generates things like delays or periodic interrupts. This is used for polling or the usleep() we have in our code.



Module: spi_usb

Purpose: The purpose of this module is to connect our USB SPI to the AXI bus. This is how the Microblaze communicates with the USB controller (MAX chip).



Design Resources

WNS	-0.686
LUT	3408
DSP	3
Memory (BRAM)	32.5
Flip-Flop	2963
Latches	0
Frequency (MHz)	93.6
Static Power (W)	0.074
Dynamic Power (W)	0.391
Total Power (W)	0.465

Conclusion

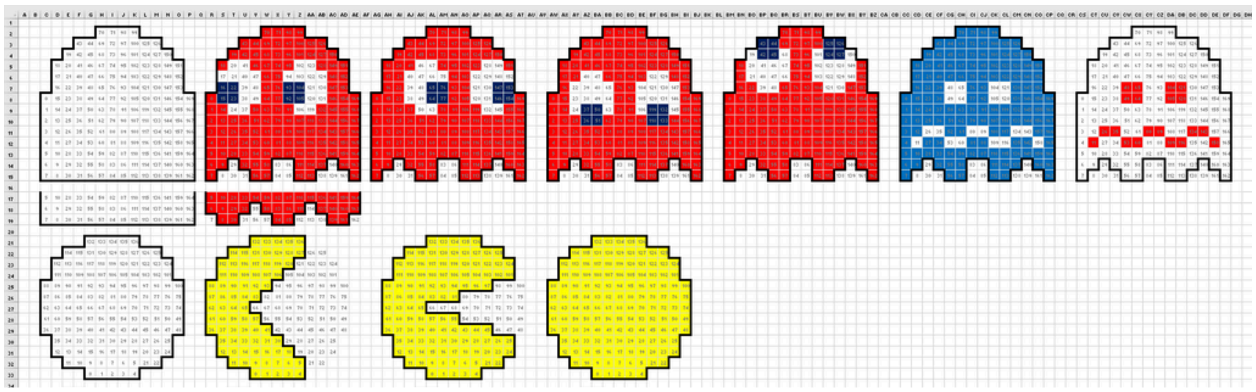
Overall this project was a ton of fun to make. We wanted to choose something that we knew would be manageable but also interesting to us. We always knew we wanted to make a game of sorts. We just couldn't decide on which one. During the first half of the project we had a couple of setbacks including needing to completely restart the project after the checkpoint. The reason we had to do this is because our approach was extremely convoluted and we realized it wasn't a plausible way to implement the game. We would have liked to incorporate true AI vs. the games “AI” however, we just simply ran out of time. If we could do this project again we definitely would choose Pac-Man again. Recreating Pac-Man was the perfect balance of challenge and enjoyment.

Appendix

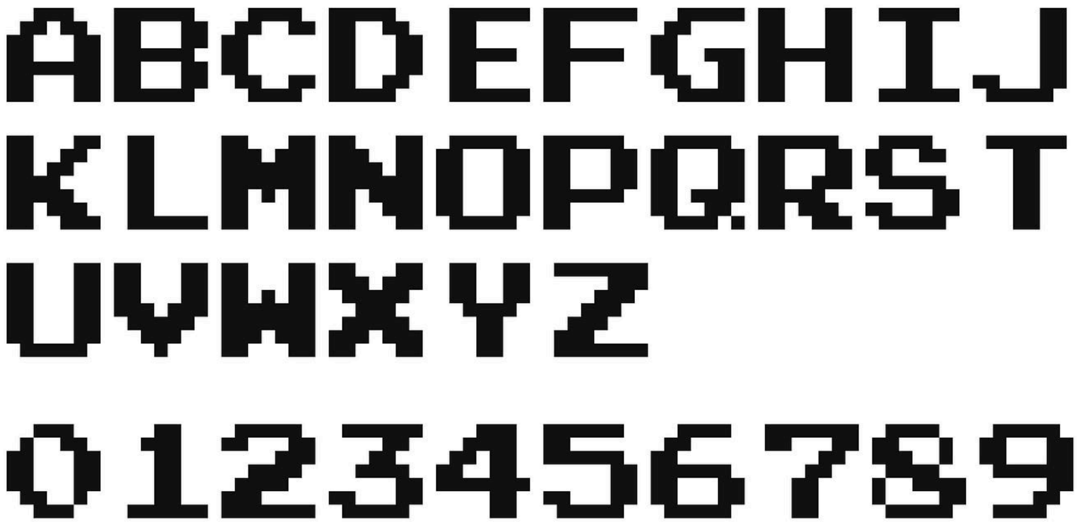
Below are images and websites that we used to figure out the specifics of sprite size, maze layout, and other resources that we used to learn about the game.

https://www.gamedeveloper.com/design/the-pac-man-dossier

Sprites



Text and Score Design



Ghost colors

Color	Color name and codes	
🟥	Name: Red Hex: #FF0000 RGB: (255, 0, 0) CMYK: (0, 100, 100, 0)	HSV: 0° 100% 100% HSL: 0° 100% 50% RAL: 3026 Pantone: 2347 C
🟪	Name: Brilliant Lavender Hex: #FFB8FF RGB: (255, 184, 255) CMYK: (0, 28, 0, 0)	HSV: 300° 28% 100% HSL: 300° 100% 86% RAL: 9003 Pantone: 2365 C
🟦	Name: Aqua Hex: #00FFFF RGB: (0, 255, 255) CMYK: (100, 0, 0, 0)	HSV: 180° 100% 100% HSL: 180° 100% 50% RAL: 5012 Pantone: 311 C
🟧	Name: Pastel Orange Hex: #FFB852 RGB: (255, 184, 82) CMYK: (0, 28, 68, 0)	HSV: 35° 68% 100% HSL: 35° 100% 66% RAL: 1017 Pantone: 1365 C