Final Lab Report

Aryan Shah (aryans5), Dev Patel (devdp2)

University of Illinois at Urbana-Champaign

ECE 385 - Digital Systems Laboratory

Professor Zuofu Cheng, T.A. Gene Lee

December 13, 2023

Introduction

Throughout the semester, we have been designing, simulating, and debugging circuits of varying difficulty using the Vivado design software and the Urbana FPGA hardware to showcase our designs. After Lab 7 came to a close, we were tasked with the outline of a final project, culminating all the prior knowledge we have gained in our previous labs into one final design. In our proposal, we pitched the idea of emulating Namco's popularized arcade game Pac-Man. For our baseline set of features, we included a maze display containing walls and borders. Collision logic such that the Pac-Man and ghosts could not go through these obstacles were to be implemented. Input via a keyboard and the necessary USB input logic would be designed so that a user could test the functionality of our game. Finally, we would incorporate the prize tokens (beans) as a method to increase score as well as ghosts that move towards the player and need to be avoided. Of course, this would all require display onto an external monitor which is why the best approach for our lab was to start with a mix of components from Lab 6.2 and Lab 7.2. We began with Lab 7.2 as it was the most complex design, but included functionality from Lab 6.2 such as the necessary GPIO blocks as well as the MAX3421E keyboard input logic code. In this report, we will provide a brief overview of some of the key features in our design, the additional hardware and software implementations designed to support it, and conclude with some additional features and next steps for our project.

Summarize the operation of the VGA-HDMI IP

As a result of choosing an arcade game for our final project, we required the necessary video and audio transmission onto the monitors in ECEB. Utilizing the same logic from previous labs, we were able to import the same VGA_controller.sv module that allowed for the creation of

timing and display signals onto a 640x480 resolution. However, to coincide with the resources we had available to us debugging outside the ECEB, it was necessary to also incorporate a VGA-HDMI IP such that we could use the HDMI video and audio transmission method. In our block design, we add the corresponding IP block that translates VGA signals into HDMI signals, noting that we still have to keep in mind our original display constraints of a 640x480 resolution and the absence of a sound module (will be discussed later).

Summarize the operation of the MicroBlaze

The Microblaze is a soft-core processor that allows for a variety of features utilized throughout this design. Firstly, it includes support for UART (Universal Asynchronous Receiver-Transmitter) which is a communication protocol that allows for serial communication between the hardware devices in our project. The MicroBlaze, additionally, supports the SPI bidirectional communication of four input/output devices on a single quad SPI bus (SPI protocol will be expanded on further). In our case, this was the method we utilized to integrate our keyboard device using USB. This allows for data transmission and receival through asynchronous communication through the SPI interface. The HDMI IP we mentioned above was also supported by the MicroBlaze processor, which includes the cores to handle low-level communication protocols that are required by HDMI. The MicroBlaze implements a necessary frame buffer to store and display the necessary video data. Specifically, this video data is simply pixel data that, after passing through our game logic, written to by our MicroBlaze and fetched by our HDMI IP from the buffer and finally displayed onto our external monitor. Finally, in order to facilitate parallel data transfer between our existing components and the MicroBlaze processor, we instantiated PIO (Parallel Input/Output) blocks in our design.

Summarize the operation of the MAX3421E and SPI Communication

To allow for the MicroBlaze to initiate SPI communication protocol with our keyboard, we had to integrate the logic of the MAX3421E, a USB host controller. In any USB device, data transfer is initiated between D- and D+ pins. These pins, at any given time frame, are at opposite binary voltage values of one another. The MAX3421E host controller C code was designed such that it will continuously initiate interrupt signals to our keyboard, which will then respond with keypress information. In this final design, our keypress information also had to be extended to include the choice between mazes (1, 2, 3) as well as the movement logic for the pacman (up, down, left, right). To expand upon the SPI protocol, it is a synchronous serial communication protocol that operates in a master-slave configuration. In this configuration, a singular master device enables the transfer of communication to and from one or more slave devices. 4 lines of communication are used, a MOSI (master out slave in to carry data from the master to slave), a MISO (master in slave out to carry data from the slave to master), a SCLK (serial clock to synchronize data transmission, and a CS (chip select for each slave device that can be enabled to indicate what devices are to be communicated within the single bus). SPI protocol is an appropriate method when prioritizing high speed and short-distance communication between devices that share a single bus.

Summarize the use of the Integrated Logic Analyzer

The ILA, integrated logic analyzer, is a debug tool in Vivado that allows us to observe and analyze various internal signals in our FPGA. Due to the complexity of integrating audio, this required us to instantiate various modules each with its own signals that we depended on to work for the audio to output correctly. During the testing stages of our design, we were faced with a

loud static noise that would only emit from one side of the headphones we used to test. It was therefore necessary to add a debug core and probe the signals to ensure they work as intended. The steps we used were similar to that of generating the debug cores in Lab 4, adding an ILA, instantiating the probes along with their widths, trigger conditions, and finally putting it all together in the top design. By being able to probe the various signals, we were able to see which signals were not working properly and alter their values in the .sv code, finally getting our audio to output properly.

Written Description of Final Project

Week 1

As per our proposed timeline, our first week of tasks included the creation of the ROM files such that the provided sprite designs can be used as separate modules. To do this, we utilized the posted link on the Discord pointing to Arnav Sheth's Image to COE file converter on Github. To begin, we drew sprites in a free online canvas drawing software. After uploading these images for the beans, "special" beans, Pac-Man, and ghosts as .JPG files, we generated a .COE file (which contains the fixed palette of color data for each sprite in a row-major order), a .py example color mapper file, .png output, .sv (output palettes), and a _rom.sv (inferred ROM initialized by the .mif). Our next step was to follow the tutorial and instantiate the necessary "Block Memory Generator" components so that we can get them stored on our on-chip memory. Additionally, another proposed progression to our timeline included the setup for our MicroBlaze and USB input. This became more complicated than expected when merging the designs for Lab 6.2 and 7.2. After taking some time to complete it, we had to modify our 6.2 logic to include the new keypresses for our USB input as well.

Week 2

As per our proposed timeline, we were to have our walls set up with the Pac-Man motion logic completed. This includes its ability to move within the maze walls as well as dying and ending the game once it comes into contact with the ghost. Firstly, we designed state values for our Pac-Man's directions. To keep the game simple, we only allowed for one keypress at a time, one direction at a time (up, down, left, right, no diagonals). After each keypress, the Pac-Man's position would be updated based on the current direction it is facing. If it reaches any of the walls, we create the same collision logic based on its current X and Y positions as well as the X and Y positions of the maze walls to not allow any overlap. While using the collision logic in the color mapper module for our Lab 6 as a base, we modified it such that instead of bouncing out, collision is detected but does not have any effect on the Pac-Man to move in the specified direction nor do any pixel values change. To coincide with the actual game, thus, the user can then continuously press a key to bump into the wall in a moment of panic but not get anywhere. We also implemented the logic to detect when Pac-Man moves over our beans/pellets. At this stage, the corresponding pellet is then removed from the screen and the score on the Hex Drivers is updated by a +1 value accordingly. Finally, we had to create logic to detect collisions between the Pac-Man and the ghosts. For now, since no special beans have been implemented yet, we only made it so that the Pac-Man coming into contact with the ghost would cause the game to end and the score to stay the same on the Hex Drivers display, indicating the final score to the user. In addition to our Pac-Man logic, we did have some concerns that we raised to our TA during the midpoint check-in regarding the feasibility of creating the maze using the same sprite conversion process. From his experience, however, he suggested a bitmap approach, using a method that stores images within a grid of individual pixels. Prior to this, we came into our

midpoint check-in with much of the collision detection logic written so in order to pivot to this design, we had to carry over this task to the next week.

Week 3 Design

We carried over the previous approach for our maze from the prior week. Due to our delayed start on this portion of the project, we agreed upon the suggested bitmap approach using C code in our Vitis application project. We had to first create our 30x80 row-column structure to instantiate onto the 640x480 display. We allocated memory for the pixel data based on the image size and color depth for our various objects. For the walls, we represented them as binary ‘0’s such that we can fill them in as a singular wall “color”, and then render the remaining hardware modules we had already designed on top of the display. Our second task for this week involved the ghost motion setup. Originally, we carried over similar logic to the ball.sv from Lab 6 but extended it such that the movements of our ghost would follow the following process: a user would keypress a movement for the Pac-Man, followed by the ghost being able to move 2 steps in its direction. However, one of our reach goals involved the integration of ‘AI’ logic into our ghosts. To do this, we instantiated yet another pathfinder IP module in our block design. This pathfinder IP would be used for the higher difficulty version of our game in which the ghost could much more easily track the Pac-Man. Utilizing the ghost navigation using estimated costs from a node (fixed position) to a goal state (Pac-Man), we were able to achieve this. The estimated costs were calculated by measuring the Manhattan distance across various paths, making it such that the ghost also avoids any collision by mazes. We explored the possibility and open-source implementations of parallelizing A* search in this design, however we had to modify a lower-level of this due to timing and knowledge related constraints.

Week 4 Design

During this last week of the project, our goal was to integrate extra finalizing components and difficulty changes to our project as much as possible. We added game start and end screens to make the game more realistic, as well as exploring around different options with mazes that can be chosen from, varying in difficulty. It is important to note that for our demo, we went with a standard maze, had one ghost using our old algorithm, one using our AI algorithm, removed all but 4 of the beans (so that the user can achieve victory without the TA having to sit through 15 minutes of gameplay), and added some special beans. The special beans logic was implemented during this week. Following similar logic for our normal beans, they were able to be eaten by the Pac-Man except they would increase the user's score by +5. Additionally, upon eating the special beans, the ghosts would turn into a red overlay and coming into contact with them would not end the game, instead they would get rid of the ghosts and increase the user's score by +5. In addition to these minor polishes to our design, most of the remaining days were spent working with audio. We decided on an SD card based approach, due to the provided files on Canvas seemingly making it easier, the discovery of a spare SD card by one of our team members, and the additional difficulty points external hardware gives along with audio itself.

Audio Integration

We wanted to emphasize the process of integrating the Pac-Man background music into our project, especially as it took longer than any other singular task in our project. To begin, we uploaded an open-source .mp3 file of the original background music onto an audio editing software. Once we looped this music a couple of times to ensure it would be long enough for a game, we exported this and then moved onto a software known as HxD. This is a hex editor

software that allows us to edit the raw data of our SD card. This means that when we exported our project from the audio editing software, we had to export its raw contents rather than another .mp3. We then had to edit the raw disk contents of the SD card and delete all of its original byte patterns, rendering the SD disk useless following this project. Once all the contents were deleted, we had to add the byte patterns from our raw audio file into the SD card using this software. Finally, after exporting this, it was time to move onto the necessary .sv modules to support audio integration. We began with the imported files from Canvas, which seemingly caused errors when trying to use them. After much deliberation on the Discord, another TA had to retract the accuracy of those files and upload a modified version days later. All in all, we had to instantiate the following modules:

SDCard_init.sv - initialized signals and configurations necessary to read audio data and manage transfer to the FPGA.

clkdiv_15Mhz.sv and clkdiv_10KHz.sv - We needed to include 2 clock dividers due to the nature of our audio signals requiring precise timing and had to be followed within the clock frequencies of our FPGA. The values selected were able to be figured out with the help of a TA.

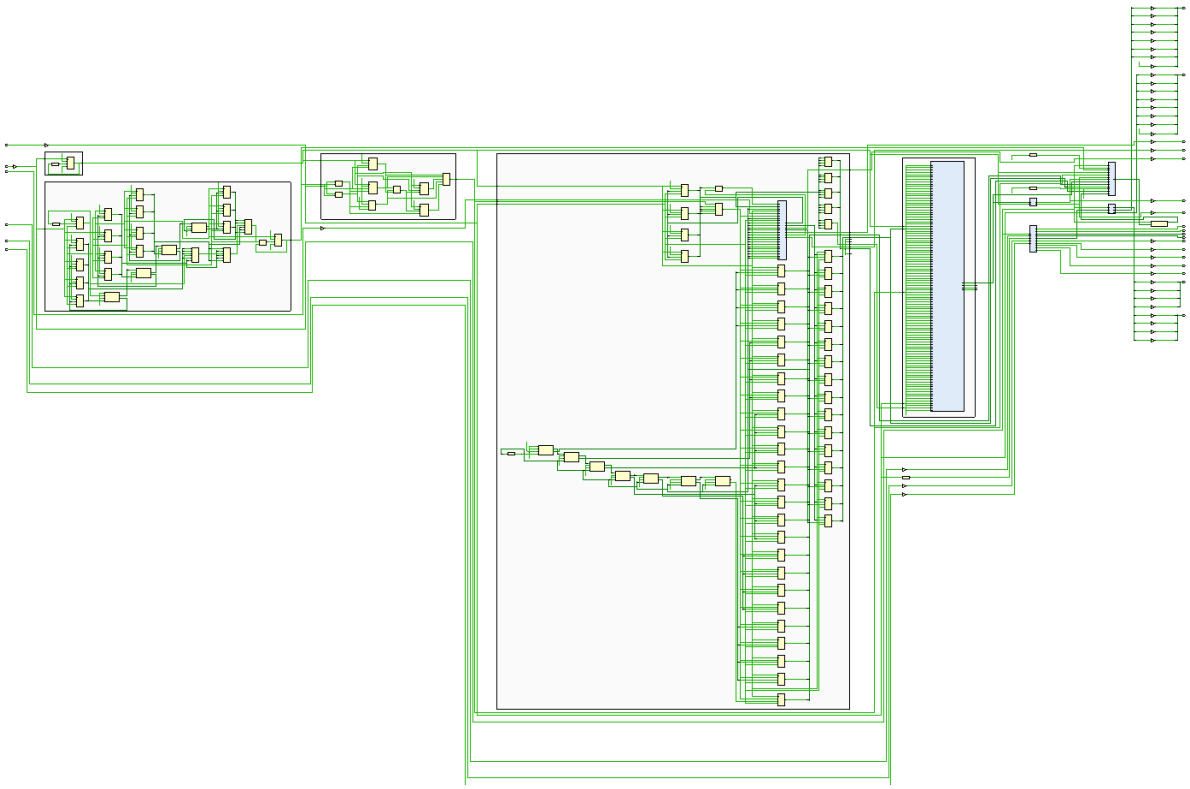
PWM.sv - Pulse width modulation encodes analog information into digital signals by approximating the analog signal from the widths of the digital pulses, changing the duty cycle as necessary to control the output of audio.

posedge_detector.sv - We had to connect our audio signals at the rising edge of each signal to coincide with the transitions necessary for synchronization.

FIFO IP module - We had to initialize a FIFO (first-in-first-out) module for the purposes of buffering our audio sample between different stages.

ILA module - We had to follow the process of our Lab 4's setup for debug core modules in order to probe different signals and generate the necessary waveforms. This was essential in our debugging stage as audio was a very complex addition that involved a lot of signals working together to achieve functionality.

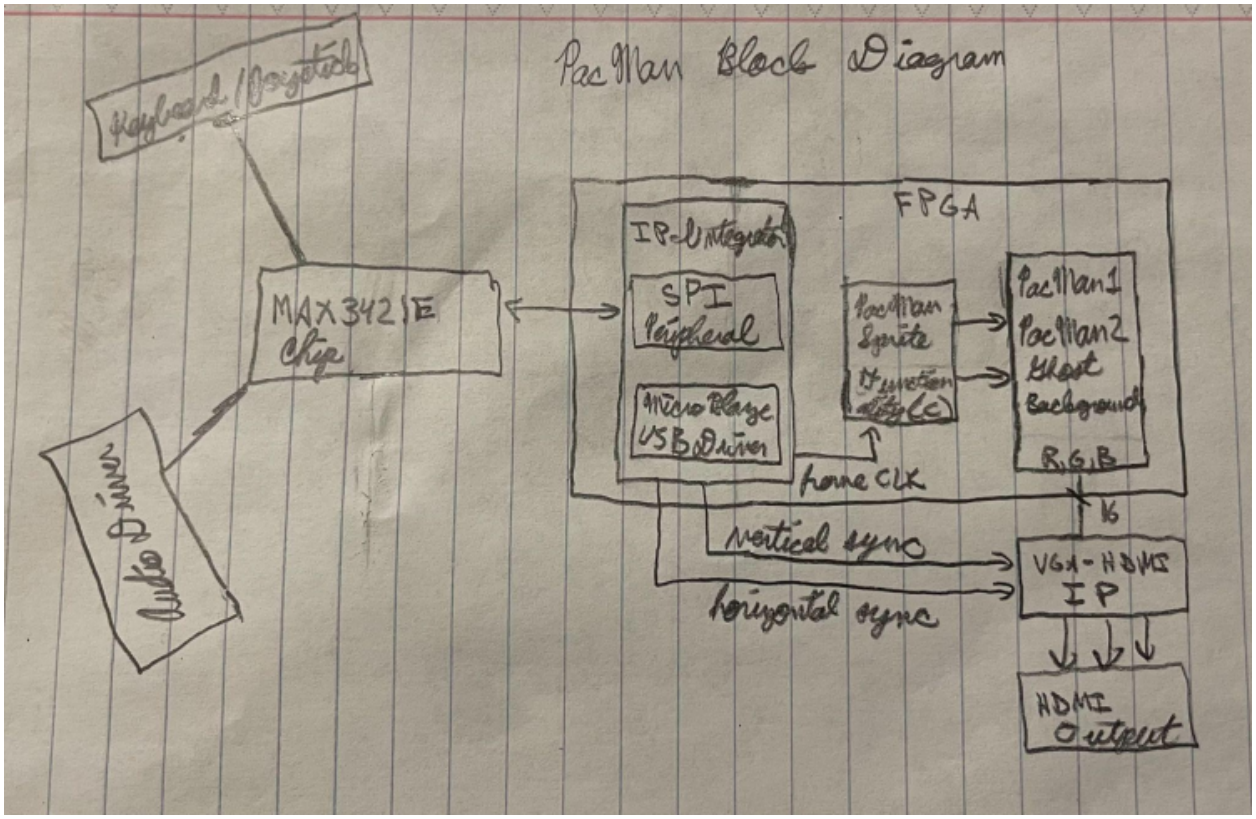
Generated RTL Viewer Block Diagram[1]



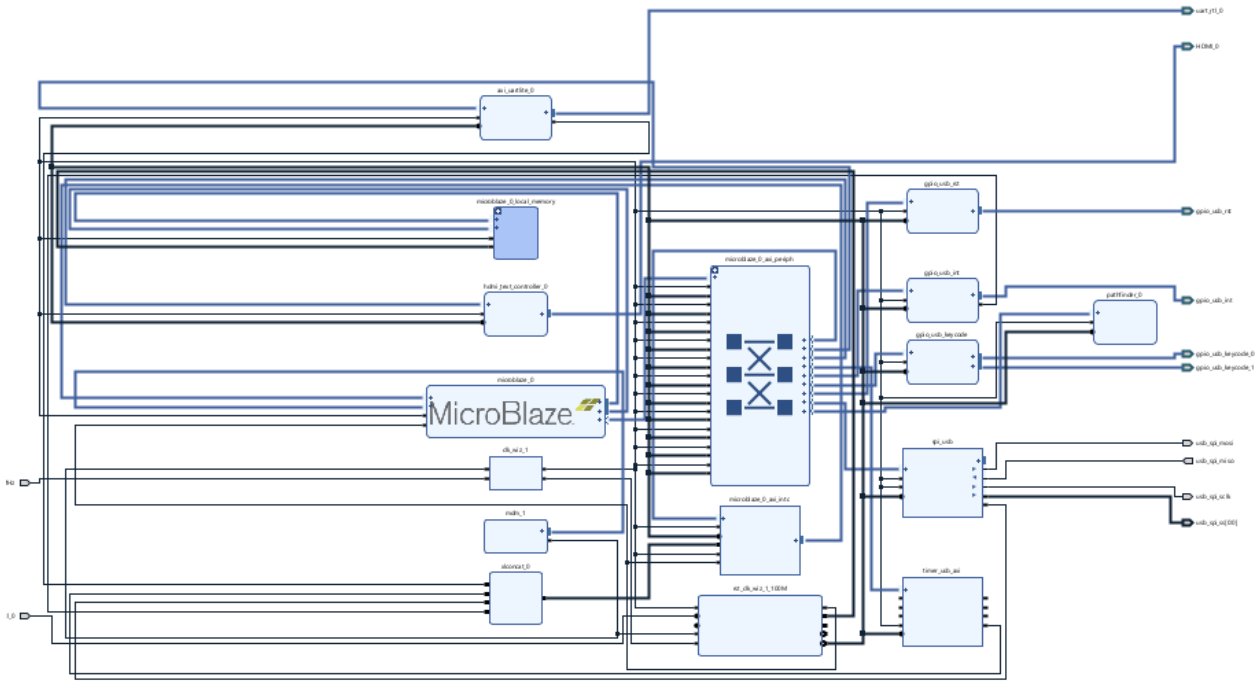
[1] Based on a Professor Cheng reply to a Campuswire post, we will not be including a waveform of our overall design as due to its complexity, any waveform output will not be meaningful to interpret

Block Diagram

Proposed Block Diagram



Implemented Design Generated Block Design



Module Descriptions[2]

Module: hdmi_text_controller_v1_0.sv

Inputs: axi_aclk, axi_aresetn, axi_awaddr, axi_awprot, axi_awvalid, axi_awready, axi_wdata, axi_wstrb, axi_wvalid, axi_wready, axi_bresp, axi_bvalid, axi_bready, axi_araddr, axi_arprot, axi_arvalid, axi_rready, axi_rdata, axi_rresp, axi_rvalid, axi_rready

Outputs: hdmi_clk_n, hdmi_clk_p, hdmi_tx_n[2:0], hdmi_tx_p[2:0]

Description: This module serves as our top level for the block design in our lab 7.2. It instantiates several components, including the AXI interface, HDMI output signals, clocking wizard, reset logic, VGA-HDMI, and vsync generator.

[2] Some of these modules remain the same from Lab 6 / 7 therefore they may be reused from prior lab reports

Purpose: Top level file, instantiates components necessary for our hardware design to interact with components such as memory-mapped AXI4 bus and VGA-HDMI output.

Module: hdmi_text_controller_v1_0_AXI.sv

Inputs: S_AXI_ACLK, S_AXI_ARESETN, S_AXI_AWADDR, S_AXI_AWPROT, S_AXI_AWVALID, S_AXI_WDATA, S_AXI_WSTRB, S_AXI_WVALID, S_AXI_BREADY, S_AXI_ARADDR, S_AXI_ARPROT, S_AXI_ARVALID, S_AXI_RREADY Outputs: S_AXI_AWREADY, S_AXI_WREADY, S_AXI_BRESP, S_AXI_BVALID, S_AXI_ARREADY, S_AXI_RDATA, S_AXI_RRESP, S_AXI_RVALID

Description: The module contains logic for the AXI read/write operations, controls values sent from the color mapper file and thus the RGB values that display onto our HDMI monitor, creates an instance of font ROM to store a sprite's data, and creates VRAM storage utilizing the BRAM we created.

Purpose: This AXI peripheral allows for communication amongst the components in our design such that we can create a text controller that allows us to render text on an HDMI screen.

Module: font_rom.sv

Inputs: addr

Outputs: data

Description: This module defines 8-bit patterns for different input addresses.

Purpose: The purpose is to create read only memory (ROM) data. Each 8x8 pattern represents a character or symbol that is to be outputted on the screen.

Module: VGA_controller.sv

Inputs: pixel_clk, reset

Outputs: hs, vs, active_nblank, sync, drawX, drawY

Description: The module generates the signals necessary for our 640x480 VGA display. The signals hc and vc are horizontal and vertical counters that check for the positions of pixels, which also tell us which lines the text is to be displayed on. It defines the logic for output sync pulse signals hs and vs, for horizontal and vertical.

Module: color_mapper.sv

Inputs: drawX, drawY

Outputs: red, green, blue

Description: The module color maps pixels based on input data. The type of character is found by the font_rom.sv file.

Purpose: Converts pixel information and input data into RGB color so that we can display our symbols and multicolor text on the HDMI display

Module: mb_usb_hdmi_top.sv

Inputs: Clk, reset_rtl_0, miso, gpio_usb_int_tri_i, uart_rtl_0_rxd, usb_spi_miso and other SPI/USB inputs

Outputs: mosi, cs, sclk, PWML, PWMR, hdmi_tmds_clk_n and other HDMI outputs, hex_segA and other hex display outputs, uart_rtl_0_txd

Description: Instantiates components within a MicroBlaze system, such as USB, SPI, HDMI, UART, audio components, and sprite components and logic.

Purpose: This module serves as the top level module for our final project design. It instantiates all the necessary components (especially those that were added to our final project and therefore not instantiated in our hdmi_text_controller_v1 packaged IP, which has a separate top-level) as well as connects them for the game logic, sound, USB SPI communication, etc. to work.

Module: Sprite_PacMan_top.sv

Inputs: sprite address, sprite read enable

Outputs: sprite data

Description: File reads data from outputted ROM files using Github converter based on the provided address. Instantiates the PacMan character.

Purpose: Provide the game with the graphical data for the PacMan Sprite.

Module: Sprite_Pellet_top.sv

Inputs: sprite address, sprite read enable

Outputs: sprite data

Description: File reads data from outputted ROM files using Github converter based on the provided address. Instantiates the Pellets sprite.

Purpose: Provide the game with the graphical data for the Pellets Sprite.

Module: Sprite_SuperPellet_top.sv

Inputs: sprite address, sprite read enable

Outputs: sprite data

Description: File reads data from outputted ROM files using Github converter based on the

provided address. Instantiates the Super Pellets sprite.

Purpose: Provide the game with the graphical data for the Super Pellets Sprite.

Module: Sprite_Ghost1_top.sv

Inputs: sprite address, sprite read enable

Outputs: sprite data

Description: File reads data from outputted ROM files using Github converter based on the provided address. Instantiates the Ghost sprite.

Purpose: Provide the game with the graphical data for the Ghost Sprite (standard difficulty).

Module: Sprite_Ghost2_top.sv

Inputs: sprite address, sprite read enable

Outputs: sprite data

Description: File reads data from outputted ROM files using Github converter based on the provided address. Instantiates another Ghost sprite.

Purpose: Provide the game with the graphical data for the Ghost Sprite (AI difficulty).

Module: PacMan_Top.sv

Inputs: movement logic for different directions, clk, reset, states for various scenarios such as pellet eaten or ghost encountered, sprite data

Outputs: display outputs, position outputs

Description: File instantiates Pac-Man sprite logic, including taking player input and moving across display, handling collision detection, movements over pellets, movements over ghosts

Purpose: Top level file for the Pac-Man sprite that takes signals from other modules such as the generated ROM data and handles logic necessary for the game.

Module: Ghost_Top.sv

Inputs: positions for the Pac-Man, clk, reset, states for various scenarios such as Pac-Man encountered

Outputs: display outputs, position outputs

Description: File instantiates ghost sprite logic, including calculating positions based on the Pac-Man sprite logic, initiates movements.

Purpose: Handles ghost logic for standard difficulty, instantiates from other given signals such as the generated ROM data.

Module: path_finder.sv

Inputs: ghost position data, pacman position data, clk, reset

Outputs: next move direction, pacman found flag

Description: Uses an algorithm similar to that of a parallelizing A* algorithm to determine the optimal path that a ghost takes to move towards the Pac-Man. Computes the shortest Manhattan distance and least costly path from the ghost's current position to the Pac-Man's position.

Purpose: Handles ghost logic for the AI level difficulty.

Module: clkdiv_15MHz.sv

Inputs: clk_in

Outputs: clk_out

Description: Divides the clock signal from the input clock by toggling the output at each instance of the positive edge of the input clock.

Purpose: To generate a divided clock signal wherein the input clock frequency is divided by two.

Module: clkdiv_10KHz.sv

Inputs: clk_in

Outputs: clk_out

Description: Counts the number of input clock cycles and toggles the output clock after a specific value, dividing the input clock frequency.

Purpose: To generate a clock with the necessary frequency (lower than the given input) to ensure the facilitation of proper audio output on the FPGA.

Module: Pulse_Module.sv

Inputs: clk, audio

Outputs: PWMR, PWML

Description: Takes incoming 16-bit audio data, splits it into left and right channel data, uses an 8-bit counter to generate PWM signals. This counter is incremented at the positive edge of the input clock. PWMR and PWML values are set based on whether the count value is less than the audio data. Upon achieving the max value xFF, the counter resets.

Purpose: Converts incoming raw audio data into PWM signals for left and right channels, allowing for stereo audio output. Accomplishes this by modulating the width of the pulses based on audio amplitude to generate analog signals from the digital signals of the audio.

Module: posedge_detector.sv

Inputs: clk, flag, reset

Outputs: sync

Description: Makes use of two flip flops that are synchronized in order to detect positive edges in a given input signal. At each positive edge, the values of the two flip-flops are updated to either the new value of the input signal or retains the value of the previous input signal. The output is assigned a value of 1 to indicate that a positive edge is detected.

Purpose: Detect positive edge transitions in an input signal, generates an output signal to determine if it is detected.

Module: sdcard_init.sv

Inputs: clk50, reset, ram_op_begun, miso_i

Outputs: ram_we, ram_address, ram_data, ram_init_error, ram_init_done, cs_bo, sclk_o, mosi_o, state_r, state_x

Description: Initializes an SD card and loads raw SD card blocks into memory. Utilizes a state machine to manage different states during the initialization and data loading process. Note that this file was imported from Canvas and Discord, provided files from staff.

Purpose: Interfaces between the SD card and our RAM (random access memory) to load raw SD card data blocks into memory. These raw blocks contain the audio data.

Design Resources and Statistics

LUT	5365
DSP	3
Memory(BRAM)	69
Flip-Flop	5466
Latches	0
Frequency	101.347927 MHz
Static Power	0.078 W
Dynamic Power	0.420 W
Total Power	0.498 W

Conclusion

Discuss functionality of your baseline design and added features.

Overall, we were able to accomplish our baseline goal of achieving the basic functionality of a Pac-Man game. We had a Pac-Man sprite that could be moved across a maze from user input. It could not go over any mazes/walls, it would die by being eaten from ghosts, and score would accumulate by eating game pellets. In addition to this baseline goal, our implementation had a variety of extra features that we had spare time to design. For instance, we had different maze game modes, as well as start and end screens for clarity within our game. Our ghosts, corresponding with the different levels of difficulty in the games, additionally included separate logic to either have a standard ghost track the user or implement an AI algorithm to track the user's movements much more accurately and find the shortest path to eat the Pac-Man. We were able to implement a continuous loop of the Pac-Man's original background audio music that can

be played on an external speaker. Lastly, we implemented special pellets that act as power-ups for the Pac-Man, giving in extra points and the ability to eat ghosts. When it comes to our proposed points breakdown, we hope to achieve a full 15/15 on functionality points during our demo. We were able to play 2 games from start to finish as well as having audio working properly during our demo. This demo tested all the basic functionalities that were written within our approved final project proposal, which include: display of walls/borders, collision detection of the walls, Pac-Man can handle USB input for movement away from ghosts, presence of ghosts that move within walls towards the player, and ability to collect prize tokens (beans). When it comes to our difficulty points, our proposal indicated a baseline functionality of 6/10. We propose the following difficulty points for our added features: +1 for the AI of the difficult ghost's movements, +1 for the use of external hardware with the SD Card, and finally +2 for audio due to it being our singular most time-intensive component that could not have been accomplished without the help of TAs.

If parts of your design didn't work, discuss what could be done to fix it.

Due to the shortened time that we had for this project, most of our components that did not work were simply scrapped and will be explained in the future steps section later. However, some functionalities of the design depicted in our demo that would like to be fixed include the switching of the Pac-Man's sprite direction when moving. We did not have enough time to import ROM files for a left, right, up, down head turn for the Pac-Man object, therefore it is simply stationary despite moving in a different direction. To fix this, we would have to instantiate new ROM files for each different image of the head turned and then implement it into our game logic to render the specified sprite when the keypress corresponding to the direction its

head should be turned is inputted. When collecting the special bean and activating our power-up, we could not figure out how to change the ghost's foreground colors to flash different colors, therefore we kept it red. Likely, it would have been a loop (terminated when the power-up effect ends) that operated at 60MHz (the refresh rate of our screen) that alternated across various RGB values to make the colors change.

What are some potential extensions of this design?

There are two specific additional features we proposed to implement for extra difficulty points earlier in our project design, but had to scrap shortly after. This was the integration of a multiplayer game as well as using external hardware input from a joystick. The idea was to have these coincide such that two users can play the game at once, one controlling a Pac-Man via keyboard and the other controlling another Pac-Man via joystick. They would work together to eat all the pellets and collect all the ghosts. This is something that we had hoped to accomplish but proved to be too time-intensive during our 4-week window, but further steps of this design should definitely aim to include this.

Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester?

We suggest that Arnav Sheth's COE converter Github link should either be included in place of, or along with Rishi's ROM converter due to its simplicity and more thorough steps for designing sprites. Furthermore, the audio files that were sent on Discord should also replace the ones posted on Canvas.