ECE 385

Fall 2024

FPGA FNAF
ONE NIGHT AT ECEB

Michael Griegel and Pratyay Gopal Reddy Rudravaram

Section XJ

Xuanbo Jin

Introduction

This report presents the development and implementation of *One Night at ECEB*, a hardware-based fan game inspired by the popular game series *Five Nights at Freddy's*. Unlike conventional game development approaches that rely on software-based rendering and processing, our game is built entirely using combinational and sequential logic circuits in hardware. The primary objective of this project was to demonstrate hardware design principles, including state machines, logic synthesis, and timing analysis, while being interactive with the user. The only portion of this project that was software based is the keyboard microblaze communication, done with SPI protocol. This report details the design process, architecture, and implementation of the game, including the logic circuits for player interactions, animatronic behavior, and game progression. Challenges encountered during the design phase and their corresponding solutions are also discussed, as well as a summary of testing and debugging efforts to ensure a seamless gaming experience.

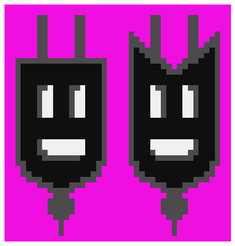
Written Description of Hardware Systems

Image to COE and ROM:

In order to include specific images as sprites in our project, we utilized Rishi's ECE 385 Helper Tools, found at the following link: https://github.com/Atrifex/ECE385-HelperTools. For each of the sprites, we first downloaded them as PNG files and then ran them through the program. For each sprite, we chose a specific color bit depth depending on the demands of the specific sprite. In most cases, we used a color palette that was 3-4 bits wide in order to get a wide range of color. Simpler sprites, such as the camera battery usage indicator, could be done in 2 bits because they were monochrome or only utilized one or two distinct colors. Each sprite also had a different resolution based on how big the image was. However, in order to stay consistent with our downsized 160x120 display, each sprite dimension was also scaled down by a factor of 4.

Once these sprites were outputted from the program, we could then instantiate BRAM as a single port ROM to hold each unique sprite. The coe files were used as memory initialization files so that the memory would be properly initialized at the start of the game and would be ready to be read from. The color palettes for each sprite were also stored as unique modules. Each specific sprite had corresponding RGB logic signals and an index signal to properly pull the right 12-bit color from memory. Each BRAM was instantiated using the negedge of the ~60 Hz vga_clk.

Sprite Transparency



Although some of our sprites, such as the background images, cover the full screen or have defined straight edges, some of our other sprites consist of irregular shapes. One such example is the Zuofu animatronic that appears in the door. Since the background of the door that he shows up in needs to be seen, we had to implement his sprite using the concept of sprite transparency. To do this, when creating the image, we used a pixel art editing tool to mask the background of the animatronic sprite with a hot pink color. When sent through the image to COE program, this color was detected as the 12-bit RGB xF0F. Then to actually account for the pink, we had to implement extra code for the drawing of the sprite: when hot pink was detected at a drawX and drawY coordinate, the sprite was not to be drawn. In this fashion, we could overlay sprites on top of other sprites without risking cutting off any important parts of the background.

Color Drawing

Although we originally believed that we would need to have a frame buffer for our display, we found that that was not the case. Our game does not rely heavily on fast screen movement or display change, so any delay that might be caused by not having a frame buffer is not noticeable. Instead, we simply used a long chain of conditionals to determine what colored pixel to draw at each positive edge of the vga clock. In order to do this, we first made a long list of different logic variables. As output to the color mapper itself, we had generic 4 bit red, green, and blue logic values. However, we also created a separate set of red, green, and blue logic values for each specific sprite/background. For example, we had an office_red signal and a cams_blue signal corresponding to the office background color and camera color respectively.

The one issue that we realized might occur with this implementation is that sprites might not appear in the correct layers on the screen. However, to completely resolve this problem, we carefully chose the ordering of our conditional statements. For example, our win screen and lose screen should completely override any other sprites that are drawn on the screen. As a result, they were the very bottom conditionals in our draw logic, meaning that no matter what came before, they would override and the rgb colors would be set for the win_rgb or lose_rgb.

We also accounted for the lights in the color selecting part of our color mapper. Using a signal “light_en” to determine when the lights were on, we simply added “1” to each of the rgb values for a sprite when light_en was logic high. As a result, since xFFF is white in our 12-bit RGB color implementation, we could easily change the color of the whole screen without much extra logic.

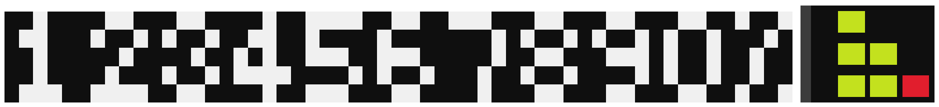
Display Scaling

Although our screen is 640x480 as defined by the VGA display output, we scaled our game down to 160x120 pixels in order to make the game take up less space and fit our intended aesthetic. To accomplish this, we simply right shifted the drawX and drawY signals by 2 bits each, effectively dividing them by 4. Then, using these new smaller x and y logics, which we called dx and dy, we could start implementing our ROM address calculations. Using row major order, we found that the 15 bit ROM address could be calculated as (dx + dy * 160). Similar calculations were also done for each of the other coordinate dependent sprites, such as the cameras and fan. These were needed because these particular sprites were implemented as multiple similar copies of themselves in order to allow for animation. As a result, we could simply select the correct dx and dy values within the sprite to choose the correct copy depending on what was happening in the game.

VGA HDMI IP and VGA controller

To implement the display as we did in lab 6, we used the same VGA to HDMI IP and VGA controller sv module that we were previously given. For our project, we utilized both the vsync and vga_clk signal outputs from the controller. Although the vga_clk was suitable for accessing the font ROMs in BRAM, major game logic, which needed to be updated on every frame, required us to use the faster vsync. Then, to allow us to output using HDMI, the VGA-to-HDMI conversion IP block came into play. This IP interfaces with the aforementioned controller using the 4-bit red, green, and blue values that come from the color mapper. The IP also needs two clock signals: a “pixel clock” to define the time that each pixel is driven into the display, and a second, faster clock (5x the pixel clock) to manage internal resources. In our case, the 640x480 VGA display requires a 25MHx pixel clock and a 125MHz 5X clock. Note the 5X clock is higher frequency than the FPGA’s 100MHz clock: the clocking wizard IP can be used to generate all required clock signals.

Battery Logic and display



During gameplay, our battery is responsible for determining how much power we have left to power the doors, lights, and camera. As a result, we created a 2-bit variable battery_state that changes value based on how many of the three power-sinks are in use at the same time. We also created an 11-bit signal batt_percent, that counts down from decimal 1584 at set increments depending on the battery_state. Decimal 1584 is the product of 99 and 16, representing the amount of battery left (starting from 99%) bit shifted by 4. This bit shift was done to allow calculations of battery drain to use integers instead of fixed or floating point values. However, for display purposes, a separate logic signal is used to represent the actual percent from 99% to 0%.

The overall game time counter updates every frame on the positive edge of the vsync clock signal. However, we didn't want to update the battery state every frame because that would lead to too much drain and the game would not be possible. As a result, we decided to update the battery every time gamecount[4:0] == 5'b00000. This allowed us to update the battery every 32 frames instead of every 1 frame, and with our variable battery drain increments of 1, 3, 5, or 9 depending on battery state, we were able to make the game difficult but still beatable.

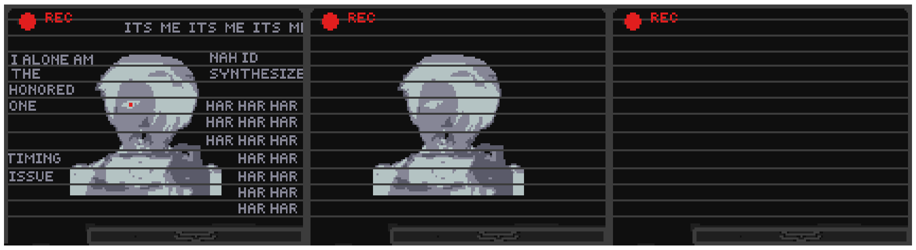
Animatronic and Jumpscare Implementation

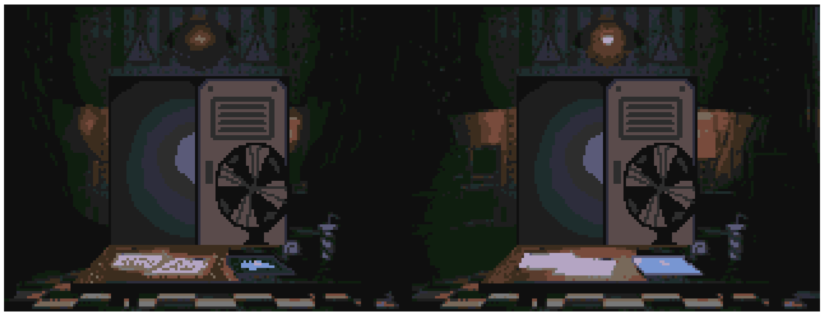
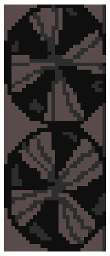
Each of the animatronics in our game appears at a pseudo-random time based on the current value of the overall game counter and a user-generated offset. In each appearance of an animatronic, the player has a certain amount of time (1 s to 3.5 s) to close the door before they automatically lose. In order to implement pseudo-randomness, the player sets a value onto the five leftmost switches of the Urbana board before starting the game. Then, the switch values are put into various logical equations in order to generate the different offsets that will be used. Although it would have been nice to make the timers completely random, in order to prevent possible collision of animatronic times, we instead used set base times with pseudo-random offsets to ensure that only one animatronic could possibly appear at one given time. If the player loses, the animatronic will turn into a jumpscare that flashes based off a separate counter time before the losing background covers the screen.

Counter logic

Although the master game clock is responsible for most of the game logic and timing, we also implemented several separate counters to allow for the timing of various other events in the game. For example, the light flickering and jumpscare flashes were both implemented using a counter to time each action. Unlike the more simple counters that we learned about and used in prior labs, each of our counters had extra logic to ensure that it would start on the correct rising or falling edge of a certain signal. For example, on each positive edge of vsync, we would set condition_delayed <= condition so that we could start counting when condition != condition_delayed. In this fashion, our counter would only start on the clock cycle after the condition signal changed values. Having separate counters allowed us to complete actions on timings that were discrete from the master game clock. It allowed us to instead use certain game logic to choose when to start an event instead of time values.

Animations





In order to make our game more lively, we created an idle animation for the fan that overlays on the office background. This animation runs off a 4 bit counter that counts up and overflows. In order to make it easier to make the animation, we stored multiple copies of the fan in the same sprite in ROM. As a result, based on the counter, we can simply just change the calculations for the correct ROM address to select between the right copy. The camera and door animations were done in similar fashions, though the camera was more challenging to implement as it required us to deal with transparent pixels once again. This is because the camera opens and closes on top of the office background; the background should still be visible until the camera is fully open.

Written Description of Software Systems

The only software component that we had for this project was the SPI keyboard code, taken directly from our code in lab 6. This code includes the following four modified functions: Maxreg_wr, Maxreg_rd, Maxbytes_wr, and Maxbytes_rd. These functions use the predefined XSpi_SetSlaveSelect() and Xpi_Transfer() Helper functions to write to the MAX3421 usb controller, and in turn, receive inputs from our keyboard. The Maxreg_wr function is used to write 1 byte of data to a specified address while the Maxreg_rd is used to read 1 byte of data from the specified address. Similarly, Maxbytes_wr is used to write multiple bytes to the address for the larger registers such as R20 or R21 and Maxbytes_rd is used to read from the aforementioned registers if the register data is more than 1 data byte long. For all four of these functions, we used a temporary write and read buffer to output and input data to the chip. The only differences between the single byte functions and the multi byte functions was the size of the read and write buffers.

The SPI protocol used is a communication protocol which operates on the principle of a Serial BUS to communicate between 2 devices, the master and the slave. The Master controls the transaction and is in charge of setting the internal synchronous clock. In total, there are 4 channels which are used in SPI: SCLK (clock), MISO (Master in Slave Out), MOSI (Master Out Slave In) and SS (Slave select/enable). Each operation is sent in packets between the devices. The first byte of every transaction is called the command byte, and it is sent by the master to the slave. This indicates the register address that will be accessed and whether the operation is a read or write. This command byte is then followed by data bytes which indicate what will actually happen to the register that we are accessing.

The SPI protocol is also full duplex, meaning that both the master and the slave can simultaneously communicate with each other. This means that the slave also sends data when the master sends data. When the master is sending the command byte, the bits that the slave sends back are called status bytes. In addition, the 0s that the slave sends while receiving the master's data bytes are called dummy bytes. When the R/W bit in the command byte is set to the read mode, the data bits arrive on the MISO instead of the MOSI and the MOSI sends dummy bits instead. The final channel, SS or slave select, is used when we have more than 1 slave per master so that we don't accidentally write to the wrong slave. In our case, our MicroBlaze was communicating only with the Max chip.

Block Diagrams

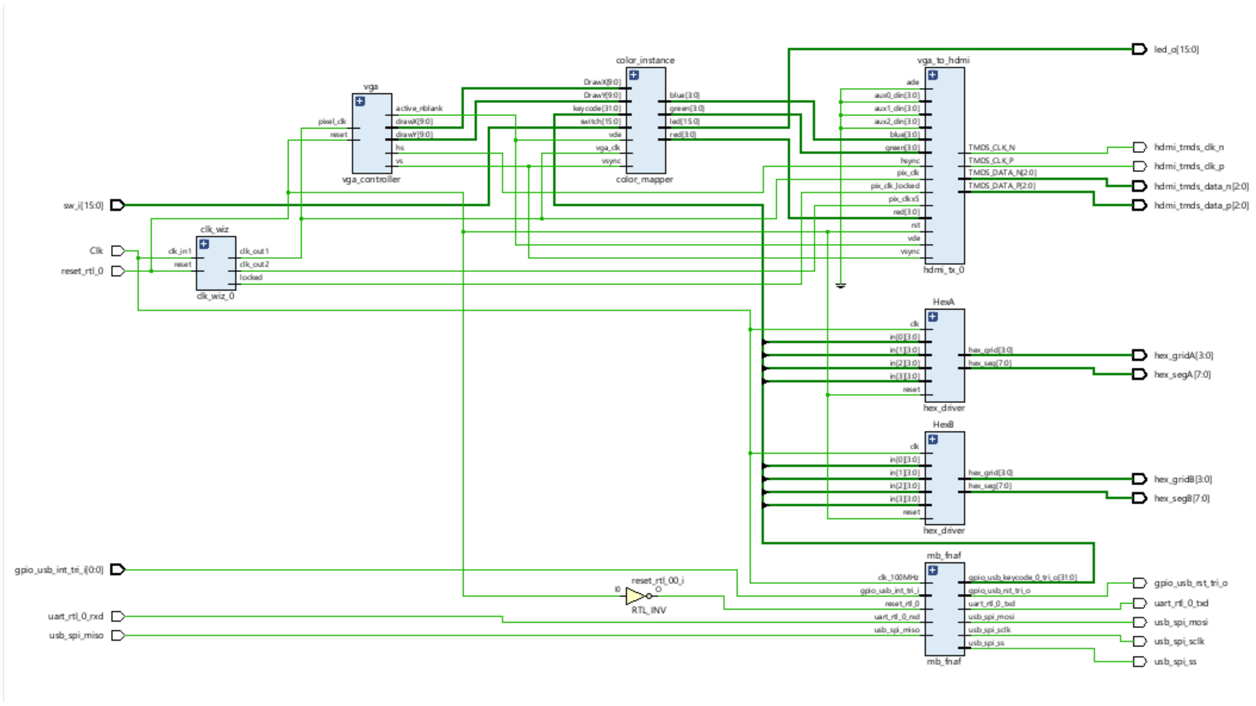


Figure 1: Overall system block design including VGA and microblaze

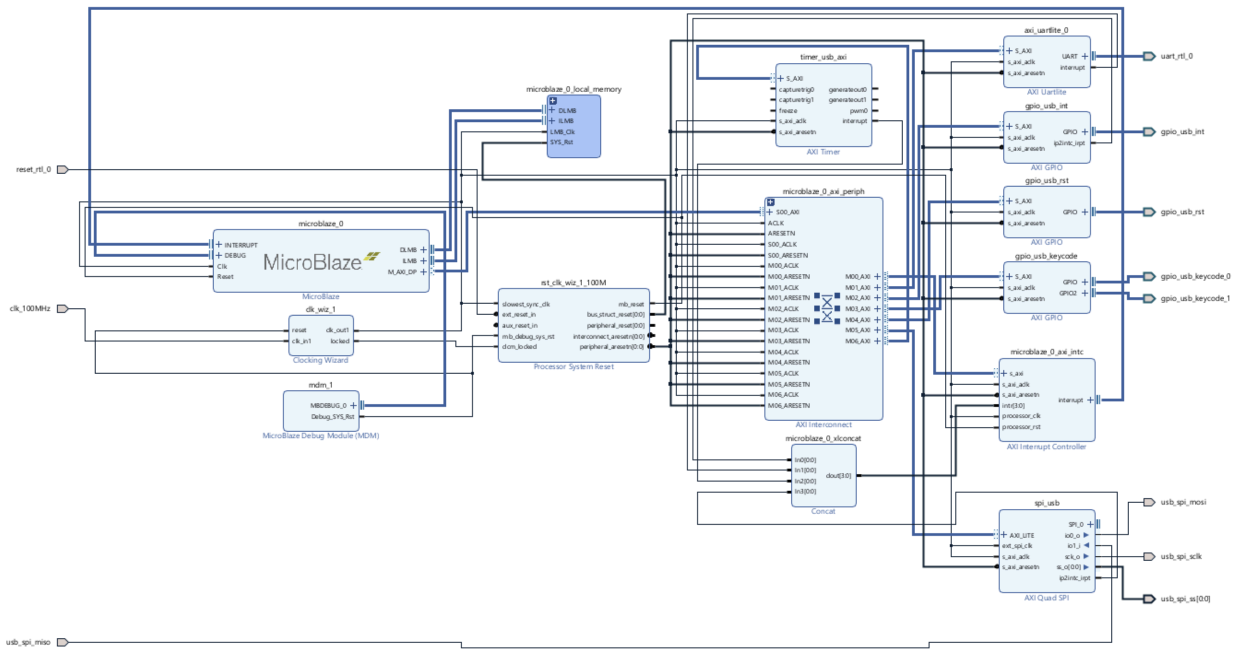


Figure 2: Overall microcontroller setup design for microblaze

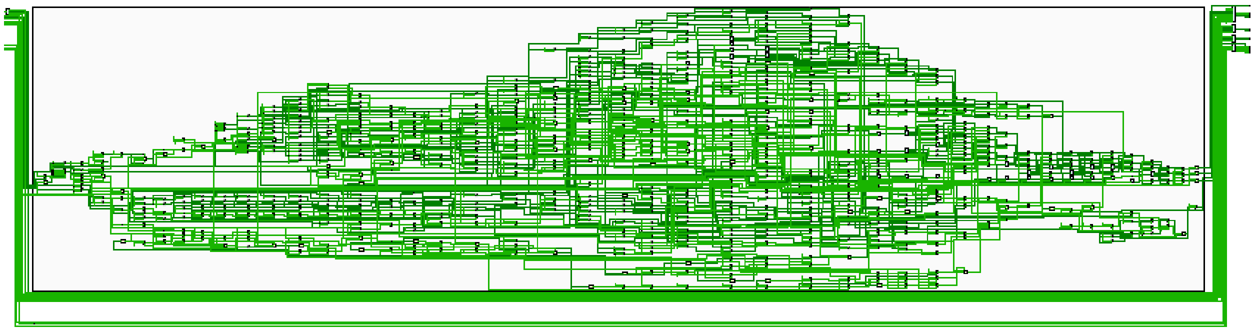


Figure 3: The whole game logic for one night at ECEB in one screenshot

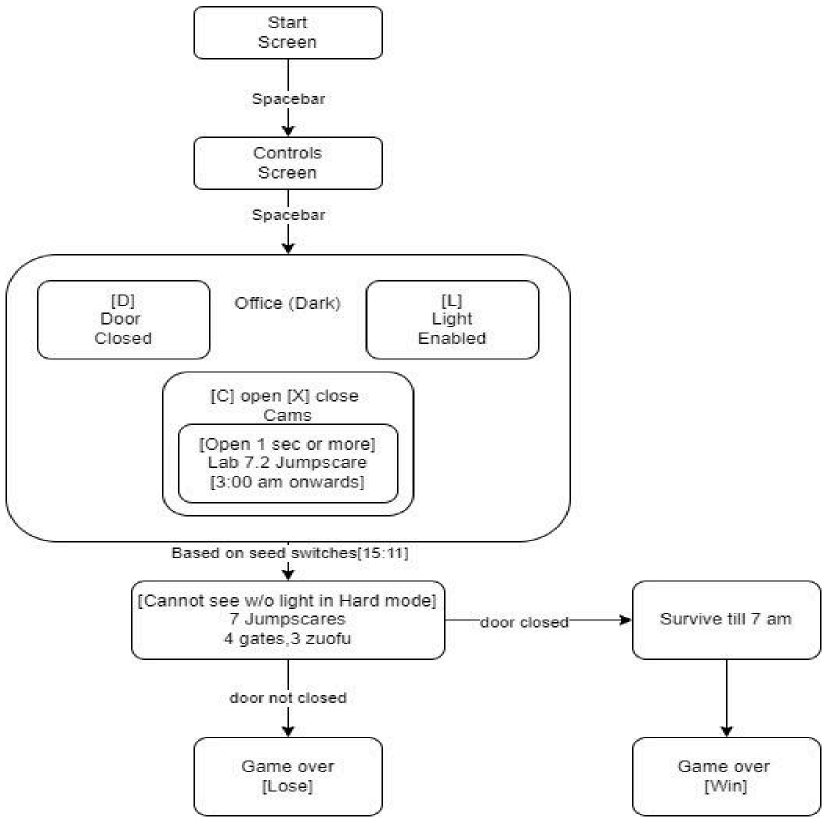


Figure 4: The game logic as a decision tree/state diagram

Module Descriptions

Module: fnaf_top.sv
Inputs: Clk, reset_rtl_0, [15:0] sw_i, [0:0] gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd
Outputs: [15:0] led_o, gpio_usb_rst_tri_o, usb_spi_mosi, usb_spi_sclk, usb_spi_ss, uart_rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0]hdmi_tmds_data_n, [2:0]hdmi_tmds_data_p, [7:0] hex_segA, [3:0] hex_gridA, [7:0] hex_segB, [3:0] hex_gridB
Description: This module contains instantiations of the hex drivers, microblaze, clocking wizard, vga controller, vga_to_hdmi converter, and color mapper.
Purpose: Top level system verilog file of the project. It is used to turn each of our individual modules into a functioning system and project.

Module: VGA_controller.sv
Inputs: pixel_clk, reset
Outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY
Description: This module implements VGA logic across an 800 x 640 sized canvas. It creates a horizontal and vertical counter that count to the width and height of the screen respectively so that the drawing cursor can seamlessly move between lines. It also creates a buffer on the right and bottom edges of the screen to facilitate this movement between lines. Finally, it creates horizontal and vertical sync pulses.
Purpose: This module sets up the logic that moves our drawX and drawY positions across the screen. It also is responsible for creating a horizontal sync pulse and a vertical sync pulse, both of which act outside the 640x480 actual screen resolution.

Module: Color_Mapper.sv
Inputs: vga_clk, vde, vsync, [15:0] switch, [9:0] DrawX, [9:0] DrawY, [31:0] keycode
Outputs: [3:0] red, [3:0] green, [3:0] blue, [15:0] led
Description: This file contains literally all of the game logic and overall implementation of the project. It initializes all of the ROMs, palettes, and contains all logic for jumpscares, game timing, and battery usage. It also handles all the color selection based on different in-game signals. It uses combinational logic to set the RGB colors for each specific pixel based on the DrawX and DrawY positions of the cursor.
Purpose: This module is what makes our game run and also makes our game colorful. It uses the drawX and drawY positions from the VGA controller to draw each pixel onto the screen during each frame. It also is what actually makes our game playable and uses lots of internal signals to coordinate the RGB outputs.

Module: palettes.sv
Inputs: [index_bit_depth - 1: 0] index,
Outputs: [3:0] red, [3:0] green, [3:0] blue
Description: This module contains color palettes corresponding to each of the sprites that are loaded into memory. Each color palette has a different bit depth depending on the individual sprite and its color needs. Each individual palette is its own unique module within this overall palettes.sv file, but for simplicity, they will all be lumped together here as they all serve the same purpose.
Purpose: This palette file is what allows us to color in the different pixels on the screen according to the sprites. It gives us 12-bit color codes corresponding to different colors present on the sprite and uses the index input to determine which color to select. The amount of bits in the index input for a specific module depends on how many colors the palette contains.

Module: COE Files
Inputs: N/A
Outputs: N/A
Description: Contains hexadecimal values relating to the data that should be loaded into memory.
Purpose: Each of our sprites was a PNG file that we turned into a COE file using Rishi's ECE 385 tool. Each of these COE files contains data about a sprite that is loaded into an instantiation of BRAM.

Module: Clocking Wizard
Inputs: reset, clk_in
Outputs: clk_out, locked
Description: Takes in an input clock signal based on the FPGA board clock cycle and outputs clock signals of other desired frequencies.
Purpose: Used in the MicroBlaze USB block design to provide a 100 MHz clock to all the various MicroBlaze components. Also used to output a 25 MHz pixel clock and 125 MHz 5x clock that are inputted into the VGA to HDMI Encoder module.

Module: HDMI/DVI Encoder (vga_to_hdmi)
Inputs: pix_clk, pix_clkx5, pix_clk_locked, rst, [3:0] red, [3:0] green, [3:0] blue, hsync, vsync, vde
Outputs: TMDS_CLK_P, TMDS_CLK_N, [2:0] TMDS_DATA_P, [2:0] TMDS_DATA_N
Description: This IP block allows a VGA controller to drive an HDMI monitor. Any VGA-compatible controller that produces HS, VS, VDE and the R, G, and B color signals (up to 8 bits each) can be used.
Purpose: This IP block is used to convert our internal VGA signal into an HDMI signal compatible with the output of the urbana board so that we can view our program on an external monitor.

Module: MicroBlaze
Inputs: INTERRUPT, DEBUG, Clk, Reset
Outputs: DLMB, ILMB, M_AXI_DP
Description: An IP based 32-bit CPU that can be programmed using a high level programming language such as C. In the microcontroller preset, it provides us an area optimized microcontroller that can be used to perform low-performance tasks.
Purpose: The MicroBlaze is the CPU that provides the basis for all the functions that we implement. It is used as the master for all the AXI and other MicroBlaze components.

Module: AXI Uartlite
Inputs: S_Axi, s_axi_aclk, s_axi_aresetn
Outputs: UART, interrupt
Description: An AXI component that allows us to interface with outside peripherals. With a set baud rate (in our case, 115200), our UART communicates with the Vitis serial terminal through the asynchronous UART protocol.
Purpose: Provides our software with printf support that can be used for debugging through the Vitis serial terminal.

Module: MicroBlaze Debug Module (MDM)
Inputs: N/A
Outputs: MBDEBUG, Debug_SYS_Rst
Description: Allows a configurable debug functionality for the MicroBlaze and Axi system. Allows the use of a UART to assist in debugging the block design.
Purpose: Used to give us the ability to debug our hardware block designs and Microblaze systems if needed.

Module: Axi Interconnect
Inputs: S00_Axi, ACLK, ARESETN, S00_ACLK, S00_ARESETN, M00_ACLK, M00_ARESETN, M01_ACLK, M01_ARESETN, M02_ACLK, M02_ARESETN, MO3_ACLK, M03_ARESETN, M04_ACLK, M04_ARESETN, M05_ACLK, M05_ARESETN, M06_ACLK, M06_ARESTN
Outputs: M00_AXI, M01_AXI, M02_AXI, M03_AXI, M04_AXI, M05_AXI, M06_AXI
Description: The Axi Interconnect provides a link between all of the AXI memory-mapped master devices and the AXI memory-mapped slave devices. It provides the 32 bit Axi bus that is used for all communication between AXI devices.
Purpose: This is the main controller for all of the other AXI peripherals and IP blocks in our block design as it generates all of the control signals for them after taking inputs from the microblaze.

Module: Processor System Reset (rst_clk_wiz)
Inputs: slowest_sync_clk, ext_reset_in, aux_reset_in, mb_debug_sys_rst, dcm_locked
Outputs: mb_reset, bus_struct_reset, peripheral_reset, interconnect_aresetn, peripheral_aresetn
Description: A module that allows the user to synchronize an asynchronous external reset input with the internal clock signal. This module also provides an output reset signal that is used as the reset signal for all peripherals in the block design such as the GPIOs.
Purpose: This module allows us to synchronize our FPGA reset button to the internal clock of the block design. It also allows us to use our reset button as an overall reset for each of the components in the block design.

Module: Axi Timer
Inputs: S_Axi, capturetrig0, capturetrig1, freeze, s_axi_aclk, s_axi_aresetn
Outputs: generateout0, generateout1, pwm0, interrupt
Description: A counter (32 bits) that generates an interrupt signal as its output. This module is capable of capturing events that trigger the timer.
Purpose: This module is largely unconnected in our project, but its output interrupt signal is one of the four interrupt signals that are concatenated together to form the overall interrupt signal.

Module: MicroBlaze Local Memory
Inputs: DLMB, ILMB, LMB_clk, SYS-RST
Outputs: N/A
Description: This is the local memory block for the MicroBlaze which provides 128Kb of storage space.
Purpose: This is the microblaze's general purpose of memory which is used to store almost everything that doesn't have its own registers. It can be extended up to 128Kb and acts as the RAM for the microblaze.

Module: Axi Interrupt Controller
Inputs: s_axi, s_axi_clk, s_axi_aresetn, processor_clk, processor_rst
Outputs: interrupt
Description: This module takes in all of the reset signals, concatenated interrupt signal, and clock signals to generate an interrupt when requested. This interrupt output then goes into the MicroBlaze.
Purpose: The interrupt controller is used to generate an interrupt signal for the MicroBlaze when it is required by the GPIO, UART, SPI, or Axi timer.

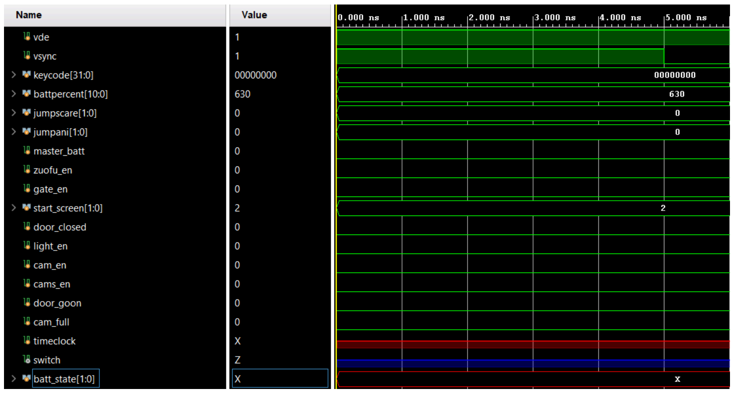
Module: AXI Quad SPI
Inputs: AXI_LITE, ext_spi_clk, s_axi_clk, s_axi_aresetn, USB_SPI_MISO
Outputs: USB_SPI_MOSI, USB_SPI_sclk, USB_SPI_SS [0:0]
Description: This module takes in the Axi clock and the output from the AXI interconnect to communicate with the MAX2431E USB controller chip.
Purpose: The Quad Spi is the main communication bus between the USB controller (slave) and the microblaze on the urbana board (master). It is what allows us to communicate the USB keyboard inputs to the MicroBlaze and other SystemVerilog modules.

Module: AXI GPIO
Inputs: S_Axi, s_axi_aclk, s_axi_aresetn
Outputs: GPIO
Description: A general purpose I/O block that allows the Axi bus to communicate with outside peripherals such as buttons, switches, and LEDs on the FPGA board.
Purpose: This microblaze component allows us to communicate between the MicroBlaze and the keyboard. It allows us to take in keycodes from the keyboard in order to drive our game logic.

Module: Concat
Inputs: In0, In1, In2, In3
Outputs: [3:0] dout
Description: Takes in interrupt signals from the Axi quad SPI, UART, GPIO, and Axi timer and outputs a concatenated 4 bit wide signal.
Purpose: Since we have 4 block components that each provide their own interrupt output signal, the concat module allows us to concatenate each of the four signals into one signal output that can then be fed into the Axi interrupt controller.

Simulation Waveform

By the demo, we had all of the signals working and we did not use a testbench as we physically exported hardware as the testbench cannot tell us the sprites being drawn to the screen. The game logic was pretty well mapped out so we did not use a testbench. The main game logic is handled by the game loop and all the following signals are used to control all of the game sprites:



Name	Value
vde	1
vsync	1
keycode[31:0]	00000000
battpercent[10:0]	630
jumpscare[1:0]	0
jumpani[1:0]	0
master_batt	0
zuofu_en	0
gate_en	0
start_screen[1:0]	2
door_closed	0
light_en	0
cam_en	0
cams_en	0
door_goon	0
cam_full	0
timeclock	X
switch	Z
batt_state[1:0]	X

<u>Design Resources and Statistics</u>

Synthesis:

LUT	1,272
DSP	8
Memory (BRAM)	0
Flip-Flop	159
Latches	0

Implementation:

WNS (ns)	1.738
LUT	3,928
DSP	11
Memory (BRAM)	60
Flip-Flop	2,932
Latches	0
Frequency (Mhz)	121.036069
Static Power (W)	0.078
Dynamic Power (W)	0.399
Total Power (W)	0.477

As can be seen in these tables, we utilized a majority of the available BRAM on the Urbana board to store all of our sprites in ROM. Our WNS is 1.738, which is not great, but good enough to run our program without any issues. We used a lot of LUTs as well, likely due to the immense amount of game logic that we made on hardware, as shown in *Figure 3*.

Conclusion

By the demo, our game was fully functional and working with all the required features as listed on the project proposal. However, we did have a slew of bugs during the design and development process that inhibited our progress. Firstly, one issue that we encountered was with the animation for camera opening and closing. Although the camera opened completely correctly, the closing animation did not work: instead, it would simply play the opening animation again. To fix this, we had to reverse the ordering of the signals in the animation to ensure that it played in reverse order.

We also had several other notable visual bugs that we had to fix. First, the way we implemented transparency for sprites caused some issues. As discussed previously, we used neon pink for any transparent pixels due to the fact that that color does not naturally appear in our project. However, although we correctly implemented our color palettes for those sprites, they were drawing incorrectly with extraneous coloring. We realized that we had simply mismatched the signals for different sprites, causing sprites to use the incorrect transparency checks that corresponded to other sprites. We also experienced some minor issues with our color palettes. When converting our sprite images to coe through the given tool, the tool incorrectly identified the same colors as different 12-bit RGB codes for different sprites. As a result, we had to tediously check each color code and make sure that they actually matched when intended. The final visual bug was also quite simple: our fan animation was clipping off part of the background. To fix this, we simply set a bound on the drawY and drawX components to ensure that the fan was not drawing outside of its circular radius.

The other bugs we had were all related to the hardware game logic. Although our jumpscares were properly occurring in most cases, we found that we could get jumpscared multiple times, even after we had already lost. In order to fix this, we had to provide default values for our jumpscare and game-lose signals to ensure that they were not floating at any point. This allowed us to purposefully set which screen would be drawn to the display at the correct moment. The final main bug we had was in our timing. To synchronize the different components of our game (clock, jumpscares, etc.) we used a counter that counted up by 60 each second. We had some minor arithmetic errors that caused our game to bug and had to match them and ensure they were correct to properly implement the animatronics and game loss/win.

The main extension that we could make from our current project is the inclusion of audio for door closing, static, and jumpscare sound effects. These audio effects could be created using BRAM to store wav files converted into coe files. We were planning on using 8 bit unsigned audio and a custom built PWM module to do this, however we ran into issues with unintended high frequency harmonics disrupting the sound. Although linear interpolation and upsampling were given as solutions to this problem, our current knowledge of sampling was not enough to understand these methods. This could definitely be something to revisit in the future if we wanted to make our game even better.