Live-Action RC Mario Kart™

6.111 Final Report

BRADLEY GROSS[1] | JONATHAN MATTHEWS[2] | NATHANIEL RODMAN[3]

ABSTRACT

Mario Kart 64 is a go-kart style racing video game created by Nintendo in which four players can each control one of eight Mario characters. The characters race in karts around a track, collecting power-ups and maneuvering to win the race. The goal of this project is to bring the classic Nintendo 64 video game to the physical world using a projector and camera, several miniature RC cars, and an Artix-7 FPGA on a Nexys 4 board. The FPGA will render an image of a two-dimensional track, which will be projected onto a flat platform from above as shown in Figure 1. The camera, mounted next to the projector, will determine the positions of the cars using IR LEDs as they race around on the platform. This camera data will be passed into the game logic on the FPGA to determine the state of the game and to control the performance of the cars accordingly. Using Nintendo 64 controllers, players will be able to collect power-ups, monitor their progress on the track, and compete with their friends.

MOTIVATION

As a group, our main motivation for this project was to recreate our favorite childhood game while utilizing as much of a Nexys 4 fpga board as possible. The Nexys 4 board is dense with a lot of inout pins, a VGA port, speakers, SD card reader and lots of block and cellular ram. Realizing all of these components capabilities, the project design evolved to be about creating a large complex system with lots of interconnects and communication between internal and external modules.

In addition to the onboard hardware, our project utilizes two OV7670 cameras, n64 controllers and external RC transmitters controlled through custom external circuits. With this many different external modules a large part of the project was about understanding how

[1] blgross@mit.edu
[2] jmatth@mit.edu
[3] naterod@mit.edu

each of these external components worked. In many cases learning our group had to learn new concepts beyond what was taught in class to control and communicate these devices. Serial data reads, bidirectional line communication and driving circuits off of externally provided clocks were all needed and utilized during implementation.

Overall our group explored a serious lesson in systems engineering and communication. The beauty of the project is shown by how these complex communication networks and controllers work seamlessly together isolating the system's complexity from the user's experience.

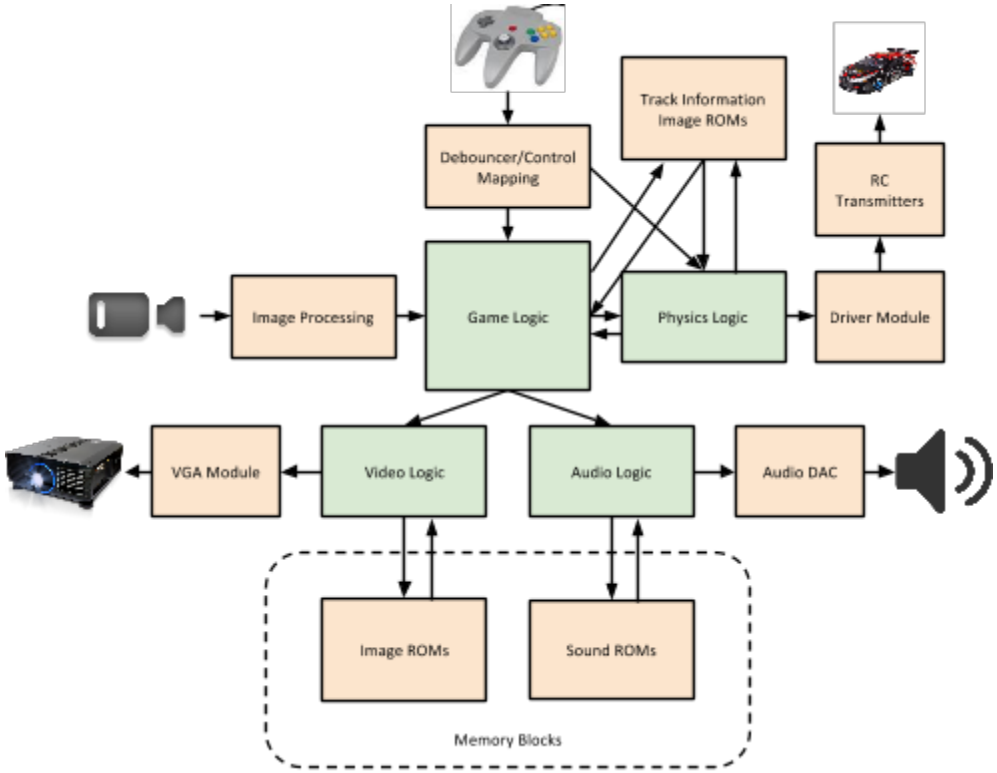
PHYSICAL SETUP

For the visual pathway of the physical system, a Sony VX-AW15 projector and two OV-7670 cameras were used to display the game and track the RC cars. A 8'x6' plywood platform was constructed, spackled for smoothness, and painted white. Two 2'x3' mirrors were epoxied onto a sheet of plywood and mounted above the platform with hinges and 2x4s. This mirrored design was chosen for two reasons: 1) so that the projector and cameras could fill the entirety of the 8'x6' area (which required a distance of 12' away based on the viewing angles), and 2) to avoid suspending the sensitive equipment above our rig.

For the car control pathway, two Nintendo 64 controllers and two "Coke Can" Miniature RC cars were used. A damaged Nintendo 64 was harvested for controller ports. The paired RC car controller transmitters were modified with transistor arrays to trigger the forward, backward, and turning pushbuttons from the FPGA. Lastly, an infrared LED was mounted on the top of each RC car, and the camera outfitted with the magnetic disk of a 3.5in floppy disk, acting as an IR bandpass filter.

SYSTEM ARCHITECTURE

The block diagram of the system can be largely divided into three overall pathways, one corresponding to each team member. These are discussed below.



RC Car Driver and Audio Pathways (Nate):

One of the goals of the project was to make it feel like the original Mario Kart game. To do this, a N64 controller was interfaced with the Nexys 4 board. The N64 has a bidirectional data line which is used to both send and receive commands. There are several different commands that can be sent to the controller. For this project, only one command is needed: to request the state of the buttons and joystick. While the game is running, the FPGA is constantly sending this command to each N64 controller via one of the Nexys 4 board's inout pins. The controller responds to each of these commands, telling the system which buttons are depressed at each instant in time.

This game uses two small RC cars that were purchased online. Each car came with a RF transmitter chip, which has four momentary push buttons. These buttons allow a user to send the commands "forward", "back", "left", and "right" to control the car accordingly. To incorporate these cars into the project, the transmitter chips were taken apart. The source and drain of a transistor was soldered across the leads of each button and gate connected to a port on the FPGA. From here the game can send drive commands to the cars based on the state of the game and the N64 controllers by driving the gates of the transistors high.

No video game is complete without cool theme music and this project is no exception. Two sound tracks from the original Mario Kart game were downloaded from the internet and converted to 8-bit WAV audio at 8 kHz. On the startup of the game, these files were loaded into the Nexys 4's on-board 16MB CellularRAM. From here, a sound player module reads samples from the memory at 8 kHz. The sound is then outputted from the board using pulse width modulation.

Car Tracking Pathway (Brad):

Vital to the game logic within the Mario Kart system is real time information on the location of the RC cars. To extract this information, the system uses the mosaiced image of two OV7671 camera's. Each of these cameras has on board A/D conversion, good frequency response in the near IR band and refresh rates of 30 or 60Hz. Taking advantage of these properties, the system uses IR LEDs fitted atop the RC cars to create distinguishable figures. Then, using an IR band pass filter, the visible light is filtered from the sensor. The filter was made using the internal magnetic disk from an old 3.5" floppy.

Having removed the visible light the cameras, the system uses a configurable threshold to remove noise and store the thresholded image in two separate two port brams with one bit per pixel. To detect the locations of the LEDs atop the cars, the system uses feedback of the LED's previous location to create a region of interest to look in. A standard center of mass accumulation is then applied around the region of interest to find the new center of mass. These centers of mass also have to be transformed into the game's coordinate system to be useful in the game logic.

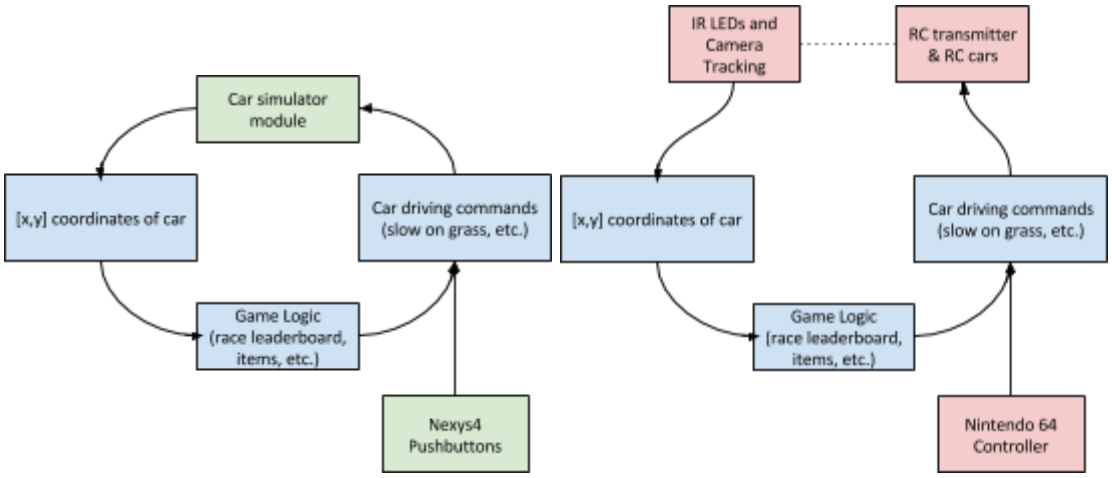
To transform the two images, several state machines were created which read the pixels from the two brams in a way which provides rotational, scaling and offset adjustments. The 90 degree rotational adjustment is done by reading the brams from a different axis than the way they were stored. This allows the system to take advantage of as much of the camera's field of view as possible. Scaling is implemented by altering the iterators over the pixel array. By skipping pixels or reading the same pixel multiple times, the system is able to adjust the tracked area to be equal to the size of the overall track. Once the images are the right size the camera images have to be calibrated to line up with the track and work seamlessly with the other camera.

To do the calibration, the directional pad and "c" controller buttons are mapped to

register offsets. These offsets are used to determine start addresses when reading from bram. The two camera's altered images are then passed into the camera tracking module so the coordinates produced can be directly used within the game logic. To calibrate, during the setup of the game, white dots on the board are used to mark where the game logic believes the car is located. Then using the controller the user can adjust the location of these circles to make the system map directly to the car.

Game and Integration Pathway (Jono):

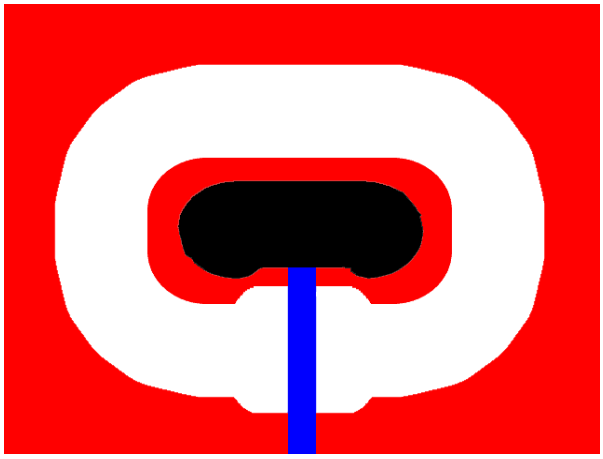
To isolate the uncertainties of the physical system of the cars and cameras, the Mario Kart game itself was developed in a closed loop. More specifically, the game was developed by connecting car driving and car tracking together in a digital car simulator module, which moved a virtual "car" and assumed perfect tracking. To integrate the vision and driving pathways discussed above, this loop would be broken, inserting the two in place of the car simulator (see Figure).



A 640x480 resolution with 8-bit RRRGGGBB was selected due to the RAM constraints of the Artix-7 FPGA. The FPGA features 4.8 Kbits of block RAM, fast enough to meet the timing requirements of VGA display protocol, but lacking sufficient space to hold more than one full-screen background image. Because of the vast amount of visual assets used for the game, an SD card was used for storage and the relevant images dynamically loaded into block RAM at the beginning of every scene. As well, due to addressing

constraints of block RAM (a 17-bit maximum address width), pixels were packed in sets of four in memory. Audio, on the other hand, has a relaxed timing requirement (the sampling rate of 8Khz gives plenty of time to read several samples from memory before output), but each clip takes up a large amount of memory. The Nexys4's on-board 16MB CellularRAM was selected to store audio, as it provided a 12.5Mhz read time with random access and large storage space. Due to the volatility of RAM, this too is dynamically loaded from SD flash memory at the start of the game.

To determine the behavior of cars on the track, a track information image was prepared that described the material of each pixel on the race track. This material information for each pixel took up 2 bits of memory — again packed into sets of four pixels due to addressing constraints — and could be one of `MATERIAL_PAVEMENT`, `MATERIAL_GRASS`, `MATERIAL_WALL`, or `MATERIAL_FINISH`, each describing the meta-properties of the track, such as the speed of a car driving on that pixel, or whether that pixel represented the finish line of the track. As well, each track information image was prepended with the coordinates of item box spawn locations on the track.



The game logic sits at the center of the Mario Kart game, listening to user input and transitioning from scene to scene, coordinating loads from the SD card. As well, the game logic keeps track of item boxes and respawn times, the items that each player owns, and the items that have been used, such as the location of bananas on the track or boosts used on a player's car. This game logic is connected to a physics logic module that performs collision detection calculations each frame, updating the game state about whether a car has hit a virtual banana, picked up an item, or completed another lap of the race. As well, the physics logic determines the operating speed, direction, and turning behavior of the cars based on their game state (e.g. on which material they are driving, any powerups that are active, etc.). In this way, the game logic can be considered as a large state machine, with the physics logic providing extra information about state transitions, such as whether a player has collided with an item box.

MODULAR DESCRIPTION

Each of the three pathways overviewed above are composed of several modules, which are discussed in detail here.

RC Car Driver and Audio Pathways (Nate):

The N64 controller interpreter modules both send and receive commands on one bidirectional line for each controller. The commands sent to controller are 8 bits long. For this project, one one command was used, 0x01, which asks the controller for the state of the buttons. The controller then responds with a 32 bit number, where the first 14 bits correspond to the state of the buttons, the next 2 bits are unused and the remaining 16 bit correspond to 2 8 bit numbers signifying the angle of the joystick. While connected to power, the controller drives the data line high. To send a command, the FPGA pulls the line low for short periods of time and allows it to return high for short periods of time. To send a 0 bit to the controller the data line is pulled low for 3 us and allowed to return high for 1 us. To send a 1 bit the data line is pulled low for 1 us and then allowed to return high for 3 us. The controller then sends bits back to the game using the timing protocol. A 1 bit sent by the controller means that the corresponding button is indeed pushed down. The N64 interpreter module sends 20 commands to the controller per second and outputs the state of all buttons.

The RC cars used for this project can receive 4 commands from the transmitter: drive forward, drive backward, turn left and turn right. There are 4 signals from the FPGA corresponding to these commands. The game uses a PWM signal on the forward and back buttons to vary the speed of the car. However, there is no information for the cars about how frequently the transmitter sends commands to the car. Because of this an ideal PWM frequency could not be determined. Ideally the car would multiple messages from the transmitter for each pwm period. Because of the unknowns in the car communication protocol, the PWM response was choppy and drained the cars' battery quickly although the desired speed modulation was achieved.

To output sounds from the board, the audio module had to read sound samples from the CellularRAM and convert them to an analog signal using PWM. The audio was sampled at 8kHz, meaning the audio logic would read from a new location in memory 8,000 times per second. To achieve a smooth sound wave via PWM, the PWM frequency was set to be 49 periods per sound sample which corresponds to 392 kHz. For each audio sample, the duty cycle of the PWM was set to the be the 8 bit sample number divided by 255 (the maximum

value of an 8-bit sample).

Car Tracking Pathway (Brad):

The car tracking system is split up into four separate modules. The first of these handles the direct Camera communication and parsing of information. The OV7670 communicates with this module with 8 data wires transmitted on the rising edge of the camera's pixel clock. The data sent from the camera is organized in an unusual pattern as the Cr and Cb bands of the image are downsampled before transmitting. The transmitted pattern sent sends the Y band for every pixel but sends the Cr and Cb bands on alternating pixel locations. After parsing and upsampling the Cr and Cb bands, the camera controller outputs a formatted data value for the pixel.

The second module within the camera system thresholds and buffers the incoming data by storing it a one bit per pixel bram. To decrease overall utilization of the scarce bram the pixels are packed into four pixel memory locations and stored based on 640 x 480 pixel location. This module uses the included bram generator within vivado which allows for two port brams with one cycle latency at 100Mhz clock speeds. Additionally a separate buffer is used for each camera.

To use the data within the bram, the third module, implemented as a series of equations, transforms the location desired from the overall game into bram memory locations. This system allows for rotations, scaling and shifting. The rotations are implemented by reading the bram vertically instead of horizontally as it is stored. Scaling is done through a simple state machine which skips pixels to downsamples when outputting or repeat pixels if upsampling. The shifting is implemented as a parameter unique to the camera that can be adjusted in real time to change the address which the module starts reading from. By locating this module between the buffer and tracker the LED tracking module is isolated from the realities of using more than one camera, camera locations relative to the board and people bumping the board or introducing noise in practice.

The last module, the LED tracker, uses a center of mass system around a region of interest. These regions of interest are set initially and then updated using the feedback of the previous LED locations. The nature of this system has a drawback however, in some cases the cars can have their positions merge into a single location and the system has no way of distinguishing the two points. This case can be avoided by tuning the ROI and calibrating the

cameras properly.

Game and Integration Pathway (Jono):

SD Controller

All assets — sounds, images, track information — reside on a single 2GB microSD card. The Nexys4 board features a microSD slot, with input/output pins available for FPGA use. A SD card controller was implemented in Verilog, based off of Steven Merrifield's VHDL implementation[4], that allows bytewise reading of 512-byte SD sectors at clock rates below 50Mhz. The SD card operates through serial commands sent through a single CMD pin and (in SPI mode) data sent through MOSI and returned through a MISO pin.

The SD card is first initialized in SPI transfer mode by sending `SDCMD0` to perform a software reset, followed by at least 80 clock cycles and `SDCMD55` and `SDCMD41` to poll for device state. Once the poll returns a ready device via the MISO line (most response types from the SD card are packaged into a single byte, the R1 type, with the first bit representing the device state and the rest containing voltage and command error codes), the SD card is ready for read commands.

The implemented SD controller module takes a 32-bit address into the SD card and sends the address as the argument of `SDCMD17` to request a sector of data. This 32-bit address must reside on a 512-byte boundary (i.e. the beginning of a data sector) to ensure proper functioning of the SD card. Several clock cycles later, the SD card responds with an R1 response, followed by a start byte of 0xFE, indicating that SD data will be sent in the next burst. The SD controller packages up the SD response data into bytes, and pulses a module output HIGH for one clock cycle when a new byte has been prepared for the video logic and other loader modules.

Because many modules need information from the SD card (the video logic into BRAM, the audio logic into CellularRAM, and track information into game state registers), a SD loader module then coordinates the loading of all active loaders. through combinational logic and signals sent to and from each loader: `do_load` and `is_loaded`.

Data for the SD card was prepared using several python scripts to extract the 8-bit color information from BMP files and 8-bit sound samples from WAV files. This data was added to the SD card using HxD, a free program that allows direct bytewise manipulation of

[4] http://stevenmerrifield.com/tools/sd.vhd

physical disks, such as that on the SD card. This ensured precise ordering and location of data on the SD card.

Video Logic

The video logic consists of several large block RAMs, each allocated to contain a particular type of image — background, player sprite, item sprite, timer, and "Latiku," a character from the Mario Kart series who counts off the start of the race with a stoplight. The SD loader coordinates the loading of these BRAMs at the start of each scene.

A scene logic submodule of the video logic determines which assets should be loaded into each BRAM for each scene. For example, the Mario Kart start screen image is loaded into the background BRAM in the first phase of the game, followed by the character select image, and then the race track image itself. The scene logic also handles animations in the game, such as flashing "PRESS START" on several screens, or animating item box selections or lightning flashes.

A dedicated timer module handles displaying a race timer during the race and lap times on a victory screen after the race has completed. This is performed by loading a timer image into BRAM and using several combinational logic blocks to determine the correct digit to display for each location on the screen. This module notifies game logic about the current race time, which is recorded on lap completions.

All image loaders have parameterized alpha removal of black color in the image, to make areas of the image transparent. This information is sent to a pixel shader module. which determines the depth ordering of pixels on the screen and any necessary alpha blending, such as with the black region around images.

No partial transparency was implemented for this project, however it is within this module that true percentage alpha blending would be achieved. The final pixel shader color output is sent to a VGA module, which determines the correct signals to send to the VGA display.

To display the same sprites at multiple locations on the screen, such as with the dynamic behavior of dropping bananas on the track, a sprite painter module was developed to

draw 10 instances of a particular sprite on the screen based on a bus flag that indicates whether that sprite instance is present (`is_present`) and, if so, its location.

In all the video logic module reads much of its data from the game logic to retrieve information such as the position of the cars, item boxes, and items, and draw them on screen. This image data is retrieved from SD flash memory and temporarily stored in block RAM at the start of each scene transition.

Game Logic

The game logic module serves as a large FSM at the center of the Mario Kart game and has over 750 individual inputs and outputs corresponding to button presses, item locations, race timers, collision detection information, and player race states. The game logic listens to button presses to navigate through the game menus and for using items. As well, the game logic contains several respawn timers for each item box, the locations of which are loaded in from the track information image at the beginning of the race from SD flash memory.

The game logic also contains several item manager submodules for each car, which handle tracking the owned items for each player, and the powerups that are currently active for each. This manager module also contains several timers for each powerup that control the duration of each effect. It is worth mentioning here that all of the game parameters, such as these item timers, but also the asset addresses on SD memory, are all macroed in a global defines file, to maximize the flexibility and consistency of the game.

Physics Logic

The physics logic effectively calculates the “next state” of the game world. This includes determining the correct drive commands to send to race cars (effecting the “next state” of the cars), and performing collision detection calculations to determine if a player has picked up an item box, or hit a banana (effecting the “state” of the player).

Drive commands are determined by examining the track information image at the car’s current [x,y] location, as well as the information image at the car’s next expected location, based on the current button presses and the calculated direction of the car’s movement from delta registers. In addition to this, the current state of item effects on each car, such as

whether a boost has been used or if a car is currently affected by the “lightning” item, is considered for the car’s speed and whether turning is inverted.

Collision detection is calculated at the four corners of the car’s driving sprite with the bounds of each item present in the world. The physics logic takes the location information of all 10 item slots (discussed in the sprite painter module above), and assesses collisions if the `is_present` flag is set.

Car Simulator

A car simulator module was developed to close the car driving and vision loop, as described earlier. This module takes forward, backward, turning, and speed inputs from the physics logic module and changes the values in a `car_x` and `car_y` register accordingly. These two registers are then passed back to the game logic, video logic, and physics logic, representing the location of the cars in the world. This simulator therefore assumes perfect tracking and driving of the cars, which was useful for developing the game in isolation of the rest of the project.

Integration

To integrate the game with the vision and driving modules, the car simulator was removed and the `car_x` and `car_y` registers replaced with wires output from the LED tracking module. As well, the forward, backward, turning, and speed signals were instead sent to the car driver module for PWM output to the car transmitters. Pushbutton signals were replaced with signals from the N64 interpreter module, opening the loop into the physical system as depicted earlier.

IMPLEMENTATION PROCESS AND REVIEW

Physical System

To incorporate all the different aspect of the project, a number of physical parts were required. A projector was needed to display the track and powerups. The surface being projected onto was constructed with plywood and 2x4s and then painted white for maximum

visibility. The platform constructed was 6ft wide by 8ft long. In order to get the projector far enough away to get that size of an image, a mirror was mounted about the track facing downward. The mirror was glued to plywood which was then mounted above the track with 2x4s and door hinges. The project was placed pointing upward in the middle of the platform. The cameras used to track the cars also looked at the mirror to get the entire track in their field of view. Mounts were designed and water jetted out of aluminum to secured the cameras in the desired orientation.

RC Car Driving and Audio Playing

The controller input module was initially implemented by itself to test the functionality before incorporating it into the system. Most of the testing was done in lab using a logic analyzer and oscilloscope to observe the waveforms on the data line. A test module used the Nexys 4's leds to display the button pushes from the controller and confirm the module's functionality.

To test the functionality of the cars and the transistor circuits used to send them commands, a test driver module was connected to the the N64 interpret module. Using the switches on the board, different PWM frequencies and duty cycles were compared to determine which signals would work best for different states of the game. For instance, a different PWM duty cycle was used for driving a car that is on the virtual grass, compared to the car that is on the the virtual road.

Testing the sound player was very straight forward. The module was implemented outside of the whole project and a small sound sample was loaded into a bram. The Nexys 4 audio output was connected to a set of computer speakers.

Car Tracking

Implementing the car tracking was made possible by incremental proof of concepts which added additional complexity to the system. To start, a lot of research had to be done on how to turn on and communicate with one of the cameras. From there some time was spent doing color correction before realizing how well the IR band pass filtering worked. Once this was discovered the design could be adjusted to use the raw YCrCb format and only retaining the Y band of the signal.

From here the next proof of concept was a single LED tracker from a single camera

using an overall center of mass calculation. The region of interest was then added along with the second LED tracker. The main complexity of the LED tracking came once calibration and a second camera was introduced. Crucial to the design, the process was split up into obtaining data from the camera, thresholding, storage, rotation and scaling and shifting and then tracking. By designing the system in this way, scaling to multiple cameras and multiple LED's was possible.

When introducing the second camera the problem shifted to being able to display a mosaiced image of two cameras. For this there was the introduction of scaling and shifting as one two camera's worth of data has to be altered to fit a vga monitor. Once the system of rotating, scaling and calibration was figured out, tracking two LEDs, the last step in the process became trivial.

In hindsight it would have been helpful to have a test rig of two constantly offset cameras to test on. Additionally, with extra time, additional care could have been used to correct minor keystoning that occurs as the cars travel further from the cameras. Lasty it would have been fun to use even more mosaiced cameras together as adding more camera's after two would be easy and from a physical perspective it looks cool.

Game and Integration

Implementing the game modules involved several proof of concept subprojects. First, a block RAM was initialized as a ROM with 8-bit RRRGGGBB image information, and the contents of the ROM drawn on the screen. This ensured that the system for displaying images from BRAM was sound, as well as the python scripts for extracting 8-bit image information from BMP files.

Next, a SD controller proof of concept was developed to read the first two bytes from the SD card, as written by the HxD program. These bytes were displayed on the 16 LEDs on the Nexys4 board. This proof of concept was arguably the most challenging aspect of developing the game modules, as a high-capacity SD card was initially used, which required a much more complicated initialization sequence and specific information about the operating conditions of the card (e.g. operating voltage). Several iterations of this module were developed before the simple 4-command implementation was

used with a standard-capacity SD card.

Combining these two proof of concepts, a third was developed that displayed the very same start screen image as in the BRAM proof of concept, except with loading the BRAM from SD card. This involved the developing of the image loader module that would become the basis of the SD loaders used for loading all image, sound, and track information data into the game, audio, and video logic modules.

From here, the modular implementation of the game was started. The working project was created with a game logic and a primitive video logic module. This facilitated moving through three scenes by using the pushbuttons on the Nexys4 board: the start screen, a character select screen, and the race screen. The shader module and SD loader were then developed to display the flashing "PRESS START" image on the start screen, the first iteration of displaying several images on one screen. The character select screen was developed next, adding on a "selected character" game state to the player. The car simulator and character sprite were added to the race track, as well as a primitive physics logic module that would move the "car" around the track. Then, the track information image was created and added to the physics logic to give the car different speeds and facilitate collisions with walls. From here, several aesthetic and functional systems were added, including the race timer, a countdown, and detecting lap completion. Lastly, a second player was added to the system before the closed loop was broken and integrate with the other parts of the project.

Along the way, issues with block RAM timing constraints and the SD card loader accumulated as more block RAMs and loaders were created. The block RAM clock speed had to be cut in half to meet timing and placement requirements and the SD controller SPI clock speed cut by 4 to meet CellularRAM timing constraints. As well, a "lookahead" register was added to the image loaders to ensure that the first pixel in the 4-pixel wide set was available in time for the VGA pixel clock. This created a smooth image across the screen, although with a small visual artifact on smaller sprites that corresponded to wrapping the first pixel of each image row to the end with the "lookahead" register.

One of the major issues with the physics logic is that the inertial properties of the RC cars have not yet been added in. As such, collisions with walls of the track are highly inaccurate, as the RC car continues to move briefly after drive commands have ceased. The car simulator in the closed loop did not simulate inertial properties, so this was not implemented, however a later iteration may change the delta registers in the physics logic to look even further ahead of the car's location, and to pulse a "backwards" signal upon collision to

simulate a sharp stop. As well, PWM improvements can be made to make the cars drive smoother around the track. This may instead be achieved by using RC cars that feature better PWM response.

CONCLUSION

This live-action RC Mario Kart racing game pushed both our team and the Nexys4 board to its limits. From assessing the optimal way to stitch two camera images together to communicating on a bidirectional wire to preparing an SD loader that met the timing constraints of BRAM and VGA display, our team faced many challenges. The system combines a complicated network of modules and interfaces with a unique physical construction to achieve an equally unique entertainment experience. While there are certainly improvements that can, and likely will, be made, the first iteration of our live-action racing video game shows in a different way what FPGAs and digital systems as a whole are capable of.