Nexys Racer

Alexander Saikalis, Shane MacFadyen, David Stamatovski, Seeyam Chowdhury
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
asaikalis@oakland.edu, shmacfadyen@oakland.edu, dstamatovski2@oakland.edu, seeyamchowdhury@oakland.edu

***Abstract*—By utilizing the multitude of available peripherals, block RAM, and clock management tiles on the Nexys A7 development board, a VGA-based racing game was able to be designed and implemented. Fundamental digital design principles were able to be applied for the project to be built from the bottom-up. State machines were used to control the flow of the game. Image data was stored in ROM for greater image detail and control. The consideration of board memory constraints and the ability to meet proper timings relative to input clocks were necessary in designing this project.**

I. Introduction

The objective of this project was to understand and apply the variety of features that the Nexys A7 board has to offer. As a development board, multiple peripherals, such as a VGA port, directional buttons, and seven segment displays, are pre-installed on the board and able to be used as input and output interfaces (I/O). Furthermore, by using an FPGA, the architecture of a project can be built with full control over the system by leveraging the ability to design and implement logic blocks from the bottom-up. Finally, the Nexys A7 is built with accessible block RAM and clock management tiles (e.g., phase-locked loop) that can be used in the design. Ultimately, these features allow a high degree of flexibility in projects that can be built.

This project, Nexys Racer, is a VGA-based video game that took advantage of the flexibility provided by this FPGA board. In the design of the project, several concepts learned in class such as block diagram design, state machine design, fundamental digital logic design, and use of peripheral I/O were applied. Additionally, due to this project's large design relative to assignments and laboratories in the class, a deeper understanding of memory instantiation, timing constraints, and project organization was necessary.

The goal of this game is for a player to control a race car and reach the finish line without losing all available lives. To add difficulty, enemy cars may be present in areas of the racetrack that the player must avoid. Furthermore, the player is able control their speed which allows for additional control of the game. Altogether, this project utilized the following peripherals on the board: the VGA port, all directional buttons, both RGB LEDs, all LEDs and both seven segment displays. Each of these peripherals are intertwined and controlled by game logic, part-specific controllers, and user input. This report provides the implementation details of how the game was designed and able to be accomplished.

II. Methodology

The design of our Nexys Racer project was split into many separate modules that could be individually tested. Each of these modules is shown in our top block diagram (attached as a separate image file). The top diagram connects each of the separate components to the outputs and other internal components.

Due to the timing requirements of the 640x480 pixel VGA display, many components of the design were run with a 25.2 MHz clock obtained by utilizing a PLL provided with the system clock of 100 MHz.

A. Main Control Circuit

1) Main FSM

FSM_Main controls a lot of the core functions of the design. It takes in data signals from collision detector, the edge detectors, distance counter, speed control and from the lives components and uses it to control the game state signal. The game state signal is the core signal that FSM_Main controls and outputs, it determines if the game is being played or paused if the player has won or lost, or if they are on the main menu. FSM main then sends this signal to the other components so then they can change their behavior based on the state if necessary. Game state is set based of the states in FSM_Main. There are 8 states, each governing what the player is interacting with.

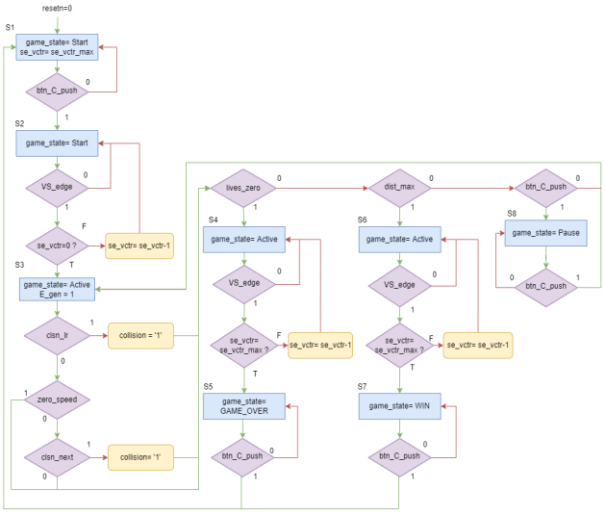

Fig. 1. Main FSM Diagram

As shown in the diagram above, state 1 and 2 both control the "start" game state with state 1 being the main menu and state 2 being a delay state. State 1 uses btn_c_push so that the user can decide when to go to state 2. State 2 takes in input from vs_edge and uses se_vctr for making a delay before moving onto state 3. State 3 outputs the game state “active” and it is the most important state because it signifies that the player is playing the game. Using the information from the other components mentioned earlier, FSM_Main determines what states are triggered next. State 3 begins with changing the game state to “active” and e_gen to 1. State 3 then uses information about collisions to determine the collision value with the signals clsn_next and clsn_lr from the collision detector. After that, in state 3 the lives_zero signal and dist_max signal are used to determine if the game has been won, lost, or is still ongoing. We enter state 4 if lives_zero ever becomes one before dist_max becomes one. If this isn’t the case and dist_max signal becomes one before lives_zero does, then the player has won leading to state 6. State 8, the state for being paused, only triggers when lives_zero is zero and dist_max is zero. It then checks for btn_c_push to be one while the game is in the “active” game state. If all 3 of those values are zero, then it stays within state 3 as the game continues. State 4 then creates a delay like state 2 before the game over, with state 5 changing the game_state to be “game_over” and then delaying to wait for btn_c_push to continue and restart the game heading back to state 1. State 6 is very similar to state 4 and state 2 with how it uses vs_edge and se_vctr as a delay. State 7 mirrors state 5 and controls continuing to the beginning of the game, waiting on the btn_c_push value to be zero. State 8 is an extra state used to control when the game is paused. State 8 loops back into state 3 after pausing is over, waiting on btn_c_push for either pausing or leaving pause.

2) VS Edge Detector

The main FSM requires a signal that sends a ‘1’ during the cycle the screen begins to refresh. Since the screen begins refreshing during the vertical sync, this input signal for the main FSM can be found by using a rising edge detector on the VS output VGA signal.

B. Button Input Management

All the inputs to the system are handled by the button input control system. The only inputs are 5 buttons, the 4 directional buttons on the d-pad and the 5th one is the center d-pad button. These button signals are all passed through a debouncer prior to any use of these signals to not cause any issues with repeated signals. The four directional d-pad buttons are passed into a small fsm that handles when the values of the signals that lead to the rest of the components are updated by the buttons. This fsm uses the collision as the update signal and this signal is from FSM Main. Once the collision signal is one it moves to state 2 where it stays until btn_en_sig becomes one. When btn_en_sig becomes one it moves back to state 1 to start updating again. Btn_en_sig is an internal signal that is controlled by a nor gate with all the btn signals fed into it. The center button (BTNC) does not get handled by this fsm, instead its lead into a rising edge detector which then converts the signal into a pulse which is now the btn_c_push signal which gets passing into FSM Main.

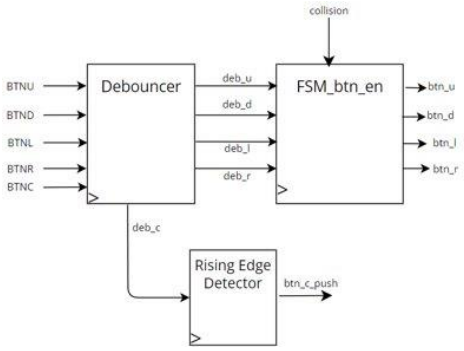

Fig. 2. Button Input Block Diagram

C. Player/Speed Control

The player_ctrl component is a register that outputs the position of the player based on several inputs. The two main inputs that control the lane position of the player are the left and right buttons. The other inputs of the player control are E, i_rst, and i_vga_frame which are used to perform logic in the player control. The player_ctrl also contains a delay counter that outputs a signal called “delay” when it reaches its max value in order to prevent the player moving immediately upon pushing a button. If the i_vga_frame, E, and delay signals are “1”, the player_pos can then be altered by user input. If the left or right buttons are selected, those respective signals will become “1” and move the player. If sclr or i_rst becomes “1”, the player_pos output will reset back to its original position. There is also a check for the bounds on the player's position so that the player does not go too far to the left or right.

The speed_ctrl component is also a register similar to the player_ctrl that controls the speed of the player based on user input. The inputs that accelerate/decelerate the speed are the up and down buttons, respectively. The enable signal is the E_gen signal that comes from FSM_Main while the synchronous clear is determined by a logical OR operation between the collision signal and start signal which both come from FSM_Main. The speed has a minimum value of zero and the max value is determined by a generic number of bits. Each button push (up or down) changes the value of the speed and, similar to the player control, there are limits so that the speed does not exceed its max or go below zero.

D. Obstacle Control

The obstacle control component controls the generation of obstacles as well as all signals related to these obstacles. It generates 6 different signals/vectors: shift enable, zero speed, obstacle positions, obstacle enables, frontwards collision, and left/right collision. The shift enable signal, E_sft, asserts a ‘1’ during every clock cycle the obstacles

shift. The zero_speed signal is set to '1' when the obstacles are not moving. The obstacle position array contains the current obstacle position for each lane, while the obstacle enable vector stores a '1' for every obstacle lane that is enabled. The frontwards and left/right collision signals contain a '1' if a collision is occurring in that direction. The following figure shows the top block diagram for the obstacle control unit.

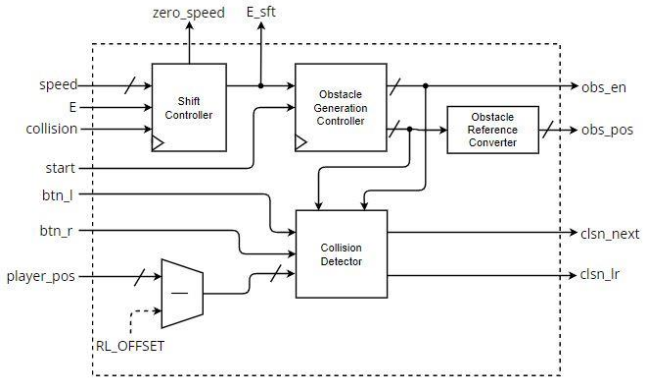

Fig. 3. Obstacle Control Block Diagram

The shift controller, obstacle generation controller, and collision detector are explained in the following sections. Since the obstacles are generated by referencing their positions from the bottom of the obstacles and the VGA controller requires that the obstacles are referenced from the top, the obstacle reference converter is needed to change the reference point of the obstacles from the bottom to the top. Due to a boundary on the right and left sides of the screen, the minimum input player position is not zero. By subtracting RL_OFFSET from the player position, the player position will then be set up for the collision detector logic. RL_OFFSET can be determined from the other constant project parameters using the following equation.

$$\text{RL\_OFFSET} = \frac{\text{DISPLAY\_WIDTH} - \text{LANE\_NUM} \times \text{LANE\_WIDTH}}{2} = 60$$

1) *Shift Control*

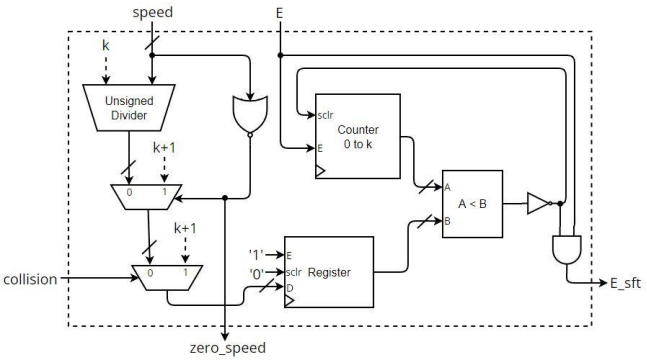

Fig. 4. Shift Control Block Diagram

The shift controller design is shown in the above figure. This circuit first uses an unsigned divider circuit to perform the operation k divided by the speed. K is a constant that can be derived using the following equation.

$$k = \frac{\text{MAX\_SPEED} \times 10^9}{\text{REFRESH\_RATE} \times \text{MAX\_SHIFTS\_PER\_CYCLE} \times \text{CLK\_PERIOD\_NS}} = 3.281 \times 10^6$$

Since k is about 3 million, the unsigned divider uses 22 bit inputs, which causes a substantial output delay. To fix this issue, most of the components used in this project are run at 25 MHz and the divider output is sent to a register that acts as a buffer in order to reduce timing issues. The multiplexors are used in order to set the register input to a value that will cause no shifting to occur when either speed is zero or a collision occurs. The output of a counter is compared to the register output. When the counter value becomes greater than or equal to the register output, the counter is reset and the shift enable signal is set to '1'. This process is infinitely repeated in order to generate the shift enable waveform.

2) *Obstacle Generation Controller*

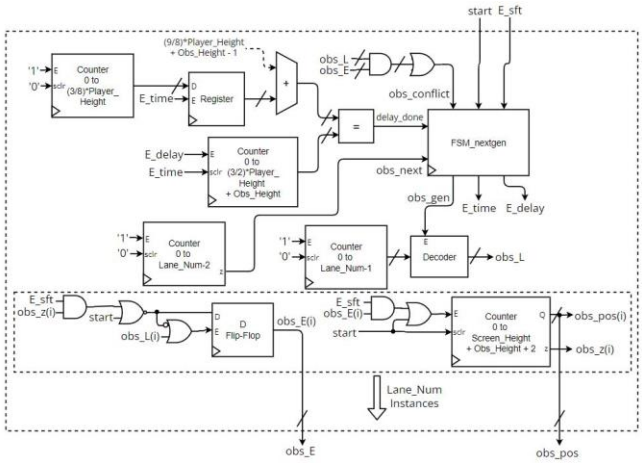

Fig. 5. Obstacle Generation Control Block Diagram

The above diagram displays the obstacle generation controller design. This design utilizes 3 free-running counters in order to pseudo-randomly generate numbers for the obstacle lanes, lane generation conflict resolution delay, and delay between obstacles.

The obstacle lane signal is input into a FSM-enabled decoder that creates the signal to instruct which obstacle to enable. After an obstacle is enabled, the obstacle position counter increases every clock cycle that E_sft is '1' until the maximum position is reached. At this point, the enable signal will be reset to '0', disabling the obstacle. The obstacle conflict signal will be set to '1' when any obstacle is being instructed to load when it is already enabled. When this occurs, the obs_next signal will be used to tell the FSM when it should try to resolve this conflict through generating a new obstacle.

The obstacle delay will be loaded into a register whenever the FSM needs to begin delaying between generating obstacles. The resulting stored value is then compared with a counter signal in order to determine if the

correct number of E_sft cycles have occurred, which is implemented through an equality comparator.

3) Obstacle Generation FSM

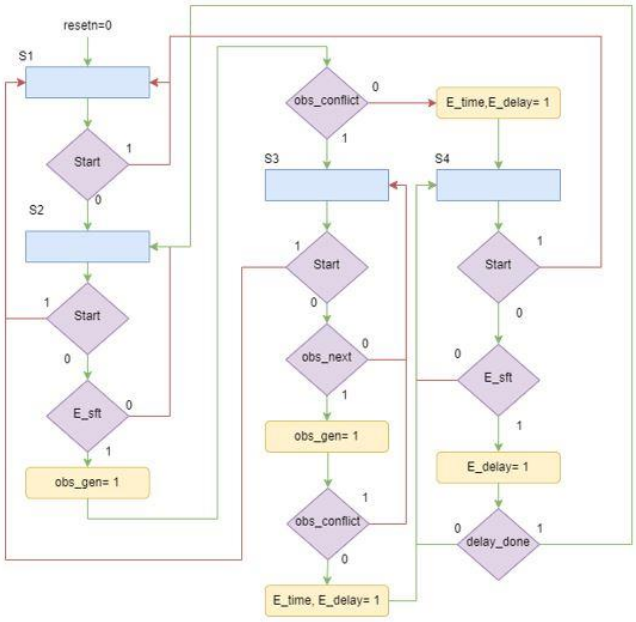

Fig. 6. Obstacle Generation FSM

The above diagram describes the FSM used for the obstacle generation control circuit. State 1 is the state where all the obstacles have been reset and the user is currently in the start menu. All other states return to this state whenever the start signal is '1'. State 2 is the obstacle generation state. After the shift enable signal is set to '1', an obstacle will be instructed to load. If there is a conflict, the FSM is sent to state 3 for obstacle conflict resolution. Otherwise, is goes to state 4 to begin waiting until another obstacle can be generated. In state 3, the FSM will attempt to load another obstacle whenever the obs_next signal is set to '1'. If there is no longer a conflict, the FSM leaves state 3 to go to state 4. In state 4, the FSM increments the delay counter every cycle E_sft is '1'. When that counter value finally becomes equal to the pseudo-randomly generated delay, the FSM returns to state 2 in order to generate another obstacle.

4) Collision Detector

The collision detector component operates based on two main process statements, the obstacle and player processes, and combines the results into the output collision signals using logic gates.

The obstacle process determines two separate vector signals using the obstacle positions and enable signals for each lane. The first vector signal, obs_next, contains a bit for each lane that will be set to '1' when the obstacle in that lane is directly in front of a potential position the player could be in. The second vector signal, obs_lr, contains a bit for each lane that will be set to '1' when the obstacle in that lane is directly to the left or right of a potential position the player could be in.

The player process first determines the next player position using the current player position, the left button input, and the right button input. Then, a vector signal, player_lane, will be used in order to store a '1' for each lane the player will occupy with their next player position.

The obs_next, obs_lr, and player_lane signals can now be logically combined in order to create the collision output signals. By using a bitwise AND operation between each obstacle process vector and the player_lane vector and then reducing the resulting vector to a single bit using OR gates, the signals for collisions in front (clsn_next) and to the sides (clsn_lr) of the player can be determined. The resulting logic circuit is shown below.

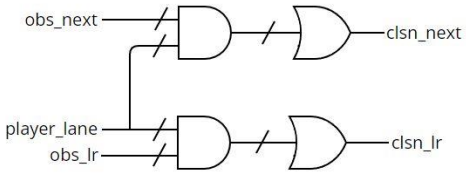

Fig. 7. Collision Determination Logic Circuit

E. Distance and Time Serializer

The serializer's purpose is to output the distance of the player's car, as well as the time the game has been running for onto the seven segment LEDs provided by the FPGA. The serializer component is also embedded with three counters: a one second counter, a distance counter, and a time counter. The inputs of this component are E_sft which comes from the obstacle control, E_gen which comes from FSM_Main, and start which also comes from FSM_Main. As suggested by its name, the one second counter is used to output a signal of "1" every second. This is done by adjusting the generic "COUNT" parameter in the counter based on the clock and the cycle time. For example, a 25 MHz clock at a cycle time of T = 10 ns, the "COUNT" should be 25*(10^6).

The distance counter outputs the distance the player has traveled for the duration of the game, holds a maximum value of 9999, and is enabled by a logical OR operation between E_sft and start. In order to feed in each digit as a separate signal to the serializer, the count of the distance (which is a std_logic_vector consisting of a generic number of bits) is converted to an integer. To break apart this integer number into each one of its digits in unsigned decimal, we use the modulus operator. The formula used for this is num = (x - (x mod y) / y, where x is the decimal number and y represents the magnitude of the place the desired digit holds. To then update the number and retrieve the next digit, we subtract the original number by its magnitude multiplied by that particular digit. For example, to retrieve the "1" in "1234" is as follows: (1234 - (1234 mod 1000)) / 1000 = 1. Now we just need to focus on 234, meaning the subtraction operation that must be performed is 1234 - (1000*1) = 234.

The digits are then converted back to a std_logic_vector. The distance counter outputs a signal "dist_max" as a "1" when reaching its maximum value.

The time counter is enabled when the one second counter or when the start signal is "1". The time counter has a maximum value of 999 seconds, and its output is also broken apart the same way the distance counter's is in order to input it to the serializer. All counters are cleared when the start signal is "1".

The serializer receives eight inputs (four from the output of both the distance and time counter) and outputs the results onto the seven segment LEDs of the board. The inputs are fed into a MUX that selects for a digit based on the state of the FSM built into the serializer. The output of the MUX is then input to the hexto7seg decoder that takes the 4-bit binary number and converts it to a value in hex from 0-9. The seven segment LEDs are turned on by a 3-to-8 decoder that also uses the same select signal from the FSM. The four left-most displays show the distance while the three right-most show the time. The fifth seven segment display was turned off to keep the two values distinct and easier to read.

F. Lives Control

The lives controller serves two purposes: first, to keep track of the number of lives that the player has available, and second, to dynamically control the RGB LEDs that are located next to the directional buttons.

The number of lives is implemented as a decreasing counter with an initial value set to three. The enable signal of the counter is the collision signal or a synchronous clear caused by restarting the game that is passed into the lives control entity. This counter provides a terminal count signal to the main FSM. When the counter reaches zero, the zero lives output is set to a logic high. Additionally, this counter keeps track of the number of lives as an internal signal.

The RGB LEDs display different effects depending on the current state of the game and the number of lives that a player has remaining. The following table summarizes these effects.

TABLE I. RGB LED EFFECTS

Game State	Lives	Left LED	Right LED
START	-	Gradient	Gradient
ACTIVE	3	Green	Green
ACTIVE	2	OFF	Green
ACTIVE	1	Yellow	Yellow
PAUSE	# Lives	Pulsing	Pulsing
WIN	-	Green	Green
GAME OVER	0	Red	Red

G. VGA Controller

The VGA controller is responsible for setting the output signals, which consist of the 12-bit RGB data, the horizontal synchronization signal, and the vertical synchronization signal to the display monitor. This controller is composed of three main aspects: a VGA driver, different screens, and a priority encoder. The VGA driver is the core component of the controller and provides signals that are crucial to what is displayed on the screen. The different screens contain data available to be drawn on the screen (e.g., a title screen or the player car). Finally, the priority encoder selects which screen(s) should be displayed on the display and is primarily a function of the current game state.

1) VGA Driver

The operating principle of the VGA protocol works around two counters that designate a horizontal and vertical coordinate (referred as the column and row henceforth) that increment on each rising edge of a pixel clock. The value of the pixel clock depends on the resolution of the display; since this project uses a 640x480 pixel resolution, our pixel clock had a frequency of approximately 25.2 MHz. It is important to note that this resolution refers only to the active region of the display. After the column and row reach their respective active maxima, the screen enters a blanking period that consists of a front porch, synchronization pulse, and a back porch (the length of each depends on the resolution). The synchronization signals provide the display a signal to indicate that a column or row is finished and to retrace to the next position. During the blanking period, the column and rows are still incrementing. For our resolution, there are a total of 800 columns and 525 rows. Once the blanking period is complete, the counters reset to zero. This process is a single frame which has a frequency of 60 Hz. Thus, our VGA driver produces 60 frames per second with 16.67 milliseconds per frame. The image below provides a visual representation of the process.

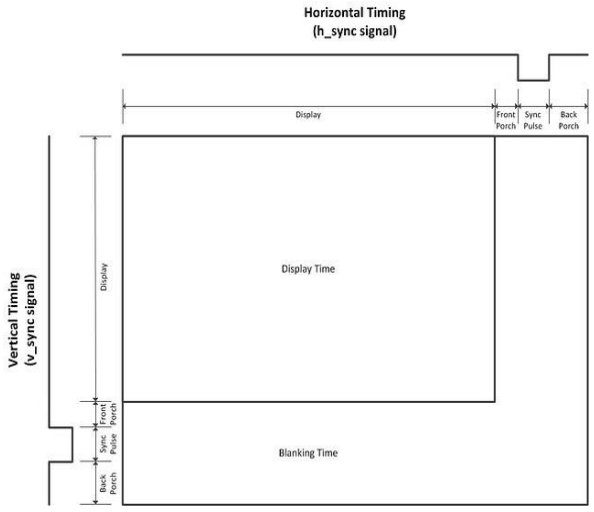

Fig. 8. Standard VGA protocol

The conventional approach to designing a VGA driver can be modeled according to the preceding logic. The difficulty, however, is that because of the sequential logic

associated with the column and row counters, there is a delay to be accounted for. Furthermore, to implement images in the project, hardware sprites stored in ROM are used. This implicates an additional delay due to ROM data elements being accessible only on the rising edges of a clock signal. These delays made drawing and positioning the sprites on the screen, specifically with column and row values of zero, impossible to accomplish.

To solve this problem, this project's VGA driver was designed differently. Instead of the blanking period being set after the active display, the blanking period was set before the display. This meant that instead of the column and row starting at zero, they started at -160 and -45, respectively. To handle the negative numbers, the column and rows are signed values. Other helpful features were added to the VGA driver which included signals that: indicate the start of each new frame, indicate the start of each new line, and indicate when the screen is active. Finally, since all the signals were registered, the column and row values had an additional register to match timings.

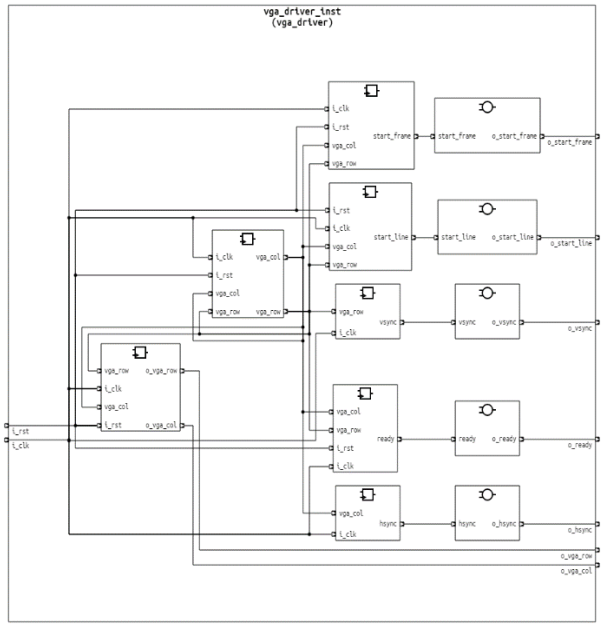

Fig. 9. VGA driver block diagram

2) *Screens*

Within the VGA controller, multiple sources of data corresponding to specific screen states were available to use depending on the state of the game. Several of these screens are designed to take up the entire screen and display discrete states of the game. These include the start (title) screen, pause screen, game over screen, and win screen. These preceding screens directly map to their respective states as dictated by the main FSM. However, when the game is active, there are several screens that are superimposed on one another, and where the priority encoder in the VGA controller becomes important. One example of how this works is the player's car appearing to be drawn over the racetrack pavement. Behind the scenes, both screens are active at the same time, with different RGB values at the same location. With the priority encoder, only the player car is selected to be drawn at the desired location on the screen.

Each screen shared a similar structure in its composition. This composition included screen specific logic (e.g., a color gradient being controlled by a timer to display as a background), a sprite drawing FSM, a sprite ROM, and a register that stored RGB data and a signal indicating if there should be anything drawn on the screen (*active*).

a) Sprites

To provide images with detail and the ability to be dynamically placed with ease, this project utilized hardware sprites. In essence, these are images that have mapped out RGB values depending on the indices of the columns and rows. Because of the size of many of these images, this project stored sprite data within ROMs. To do this, a script was written in Python that reads the RGB data of each pixel within an image, for any given resolution, and then outputs a VHDL file that is structured such that Vivado's synthesis tools can infer it as ROM. Correct inference occurs when there is a data array (prefilled with values) that can be accessed with specific element (address) values on each rising edge of the clock.

Some of the images, such as those containing text, were created with image creation tools, such as Affinity Designer. The caveat with creating images this way was that dots per linear inch (DPI) and RGB fidelity are both different in modern displays when compared to 12-bit color VGA display available from the board. There was a loss of quality between the image and the final product- particularly if any gradients were used. For this reason, objects such as the player and enemy cars were drawn with a tool specifically meant for pixel art called Piskel. The figures shown below provides an example of a logo and the player's car.

Fig. 10. Paused logo made using Affinity Designer

Fig. 11. Player sprite made using Piskel

b) Sprite ROM

Each image had a unique ROM file. After inference, Vivado placed these files into the board's various ROM blocks using the following depths (measured in bits): 8,192,

16,384, and 32,768. The depth that the sprite requires is dictated by the product of its width and height. This was an important detail to consider since Vivado would place a sprite with a depth of 8,193 into a ROM with 16,384 which not only increases the time to synthesize but also increases ROM utilization on the board. The ROM's addresses provide RGB data starting from the top left of the sprite down to the bottom right of the sprite. The image below shows an example of a ROM block diagram.

Fig. 12. Player car ROM block

Altogether, the multitude of ROM inferences can be seen in Vivado's synthesis report for the project as shown below.

ROM: Preliminary Mapping Report

Module Name	RTL Object	Depth x Width	Implemented As
start_screen	gamelogo_sprite_inst/sprite_addr_reg_rep	32768x10	Block RAM
start_screen	button_sprite_inst/sprite_addr_reg_rep	8192x12	Block RAM
start_screen	button_sprite_inst/sprite_addr_reg_rep	8192x12	Block RAM
pause_screen	paused_sprite_inst/sprite_addr_reg_rep	16384x12	Block RAM
pause_screen	button_sprite_inst/sprite_addr_reg_rep	8192x12	Block RAM
pause_screen	button_sprite_inst/sprite_addr_reg_rep	8192x12	Block RAM
gameover_screen	gameover_sprite_inst/sprite_addr_reg_rep	16384x12	Block RAM
gameover_screen	button_sprite_inst/sprite_addr_reg_rep	16384x4	Block RAM
win_screen	winner_sprite_inst/sprite_addr_reg_rep	16384x4	Block RAM
win_screen	button_sprite_inst/sprite_addr_reg_rep	16384x4	Block RAM
collision	explosion_sprite_inst/sprite_addr_reg_rep	8192x8	Block RAM
player	player_sprite_inst/sprite_addr_reg_rep	8192x12	Block RAM
enemy	enemy_sprite_inst_0/sprite_addr_reg_rep	8192x8	Block RAM
enemy	enemy_sprite_inst_1/sprite_addr_reg_rep	8192x8	Block RAM
enemy	enemy_sprite_inst_2/sprite_addr_reg_rep	8192x8	Block RAM
enemy	enemy_sprite_inst_3/sprite_addr_reg_rep	8192x8	Block RAM
enemy	enemy_sprite_inst_4/sprite_addr_reg_rep	8192x8	Block RAM
track_border	right_track_sprite_inst/sprite_addr_reg_rep	8192x5	Block RAM
track_border	left_track_sprite_inst/sprite_addr_reg_rep	8192x5	Block RAM
pavement	finish_sprite_inst/sprite_addr_reg_rep	8192x1	Block RAM

Fig. 13. Vivado block RAM synthesis mapping

c) *Sprite FSM*

While there were multiple ways of drawing the sprite on the screen, the most convenient way to do so ended up being to utilize a state machine. The sprite FSM mirrors the way the VGA driver increments the display column and row indices. For the state machine, the sprite column and row are analogous to the display coordinates; the only difference is that the sprites' dimensions are smaller. The crucial signals provided by the FSM which are used by higher order entities are the sprite address and an active signal.

Importantly, the FSM is easy to use as it requires only two key input signals. First, there is a start signal. The logic of this signal is set externally within the screen specific logic of each screen. Start should be active high when the VGA display column and row are equal to where the sprite should begin. Second, there is a starting column value (which must be signed) that determines the value of the column the sprite should begin drawing at.

The different states are as follows. First, when nothing is being drawn, the state remains in *IDLE*. Second, when the start signal becomes active high, the FSM's state switches to *START*. At this point, the registered values of the sprite row and sprite address are set to zero. The state now becomes *AWAIT* which signifies the start of each line in the sprite. As a result, the registered sprite column is set to zero. When the VGA column is equal to the start column minus two (which accounts for delays caused by multiple clocked values), the state switches to *DRAW*. During this state the active signal is high and the sprite column index increments. Provided that the sprite column and rows are within proper bounds to ensure a correct increase of position, the sprite address also increments. If the sprite column is not in its final column (determined by the sprite's width), the *DRAW* state will remain persist to draw one line of the sprite. While in this state, the active signal is set to logic high. After the final column, the state machine checks if sprite row index is in its final row (determined by sprite height). If it's not, the state switches to *NEXT LINE*, increments the sprite row, then switches the state back to *AWAIT*. If it is the final row, the sprite is finished drawing and the state is set to *DONE* before switching the state back to *IDLE*. The state machine can be visualized in the image below.

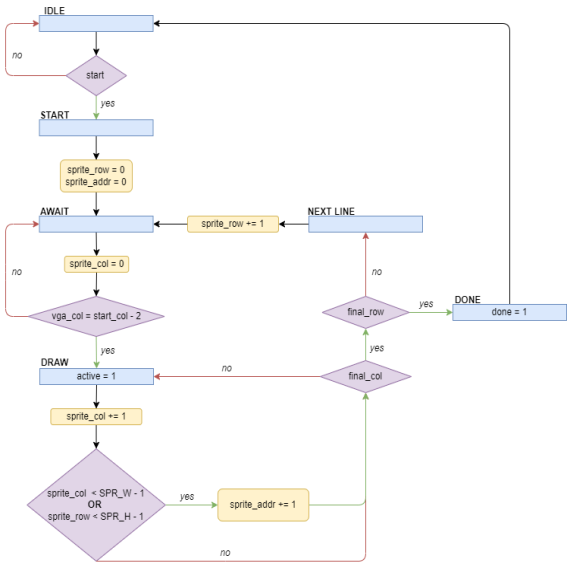

Fig. 14. Sprite FSM diagram

d) *Screen Registers*

The interaction between the screen logic, sprite FSM, and sprite ROM allows for each screen to display the sprites according to the desired behavior. The screen logic dictates when and where the sprite should be drawn, the sprite FSM provides addresses to the sprite ROM and supplies an active signal, and the sprite ROM provides the RGB data of the from the supplied address. For sprites that have motion, an additional registered process was included to shift the start position at each *vga_frame* signal to ensure smooth motion. Finally, the RGB data and active signals are registered. The register also contains priority encoded logic to select which data should be stored (if there are multiple data sources). An example of a more complex module is shown below.

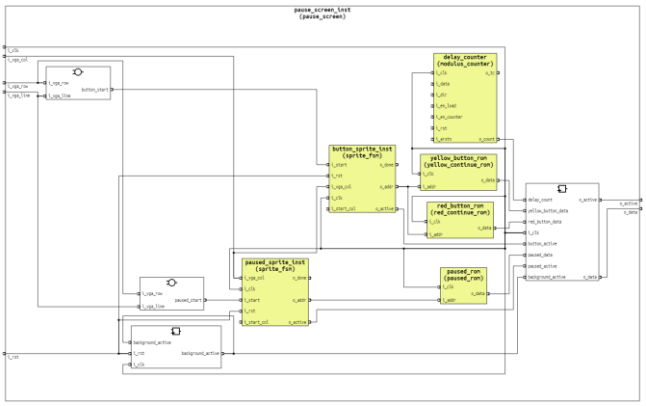

Fig. 15. Paused screen block diagram

3) *Priority Encoder*

Since all the screens are always active, it was necessary to select which screen(s) should be enabled at a given time. To do this a combinatorial process exists in the VGA controller that properly selects what should be displayed. Conditional statements that occur higher up in the conditional chain have priority in the display. Many of the basic state screens (e.g., win screen) can only occur when the game state is on a particular state. Nothing else is displayed at that time. On the other hand, the desired display might be multiple screens superimposed on one another. For the pause screen, not only is the paused dialogue on display, but also the entirety of the paused racecars and racetrack in the background. To accomplish this, the priority encoder will draw the screens on top of one another according to the conditional hierarchy. This culminates in the following structure of the VGA controller.

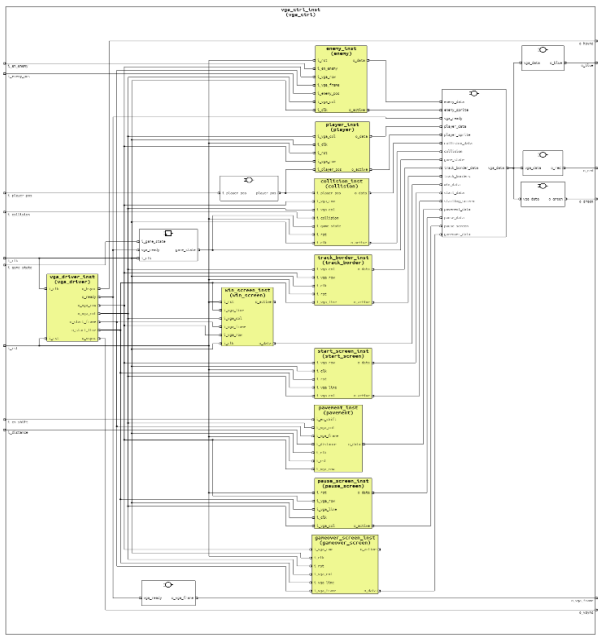

Fig. 16. VGA controller block diagram

III. Experimental Setup

First, simulations were run on many of the individual components to ensure they operated as expected. For some of the components, the constant parameters were scaled down to reduce simulation time. The resulting simulation testbenches were verified by observing various internal and output signals to confirm they responded correctly to the chosen input stimuli. Simulations were run on the main FSM, player control, speed control, obstacle control, and VGA control VHDL files to ensure they worked correctly.

After combining all the files into a single top file, it was very difficult to confirm the circuit was operating correctly from viewing the simulation since many of the output signals are hard to interpret correctly. Therefore, the top file for our project was tested by implementing the design on the board and viewing the screen on an external monitor connected by VGA. This allowed us to visualize the output of our design and ensure proper functionality.

IV. Results

The Nexys Racer game performed as intended regarding the various game screens, button inputs, and the overall gameplay. The testbenches and simulations were created to make sure each component functions in the way it was intended and showed waveforms that confirmed the components were working. As far as issues in the final version are concerned, there were none to take note of which was a positive sign. Topics we covered in class about the datapath and control unit, embedded counters, and the serializer aided the process in creating our project tremendously.

Conclusions

Working on this project allowed us to gain deeper insight into the complexities and challenges when working with VGA. Coupled along with our already intricate design, it made staying on task and scheduling weekly meetings a priority to stay on track and complete the project on time. One main take-away point from this project is the need to find information outside of classroom resources and to learn it on our own to be able to use it.

To improve our Nexys Racer game, extra features could have been implemented to improve player experience and the overall quality of the game. These include different game modes, sound effects, external peripherals, etc.

References

[1] D. Llamocca, “VHDL Coding for FPGAs,” Reconfigurable Computing Research Laboratory (RECRLab), [Online]. http://www.secs.oakland.edu/~llamocca/VHDLforFPGAs.html.

[2] S. Larson, “VGA controller (VHDL),” Digi-Key, 17-Mar-2021. [Online]. https://forum.digikey.com/t/vga-controller-vhdl/12794.

[3] W. Flux, “Hardware Sprites,” Project F - FPGA Development, 04-Feb-2022. [Online]. https://projectf.io/posts/hardware-sprites