Lab 4 Project Report

Harrison Cassar, Jasper Edbrooke, Eldon Ngo

Lab Section 1, CS M152A

TA: Rahul Garg

Designing and Implementing a Dinosaur Game that Resembles Google Chrome's T-Rex No-Internet Connection Game

Harrison Cassar
2nd-year CS

Jasper Edbrooke
3rd-year CS

Eldon Ngo
2nd-year CS

Problem Statement and Design Description

Throughout this entire project, since no code was provided to us like in previous labs, as well as no specification or external requirements (all requirements were internal, being determined by us both before and during the implementation of our project), all source code—with the exception of the main VGA module, as explained later in the report—was developed by our group in its entirety. This lab serves as a review and implementation of the design techniques we learned as a group through the completion of the previous labs.

For this project, our group decided to implement our own version of Google's T-Rex game featured on the Google Chrome browser, attempting to as closely as possible mirror the functionalities presented in the game (as depicted in Figure 1). As an important note, the main focus of our project was to mimic the functionalities—not the graphics—of Google Chrome's dinosaur game, as we believe the main value of the game comes directly from how the game plays and not exactly how it looks. Figure 2 presents a visual representation of what our version of the game looks like, where the dinosaur, cacti, and pterodactyl sprites are replaced with appropriately-colored and shaped simple representative rectangles.

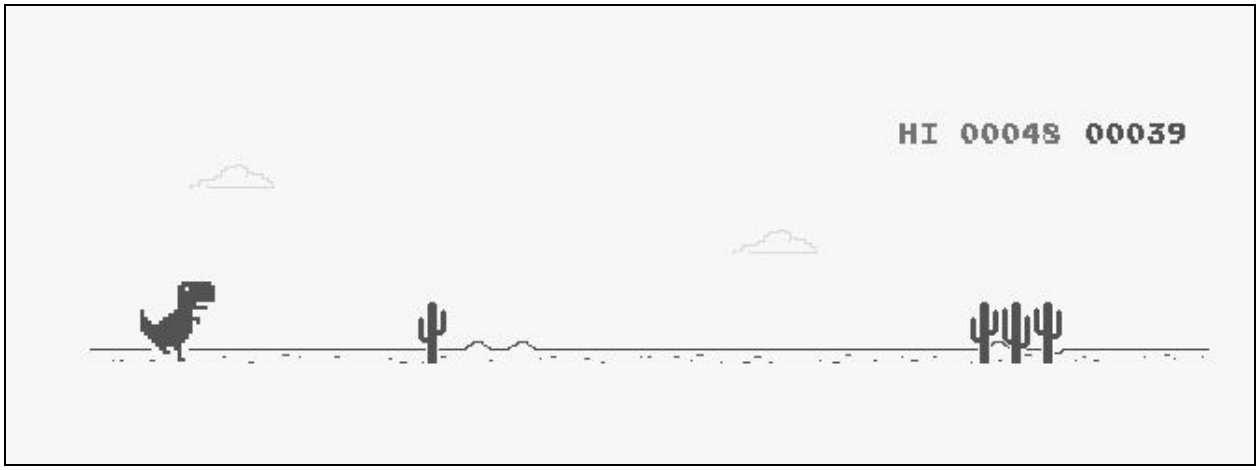

Figure 1: Screenshot of Google Chrome's T-Rex Game

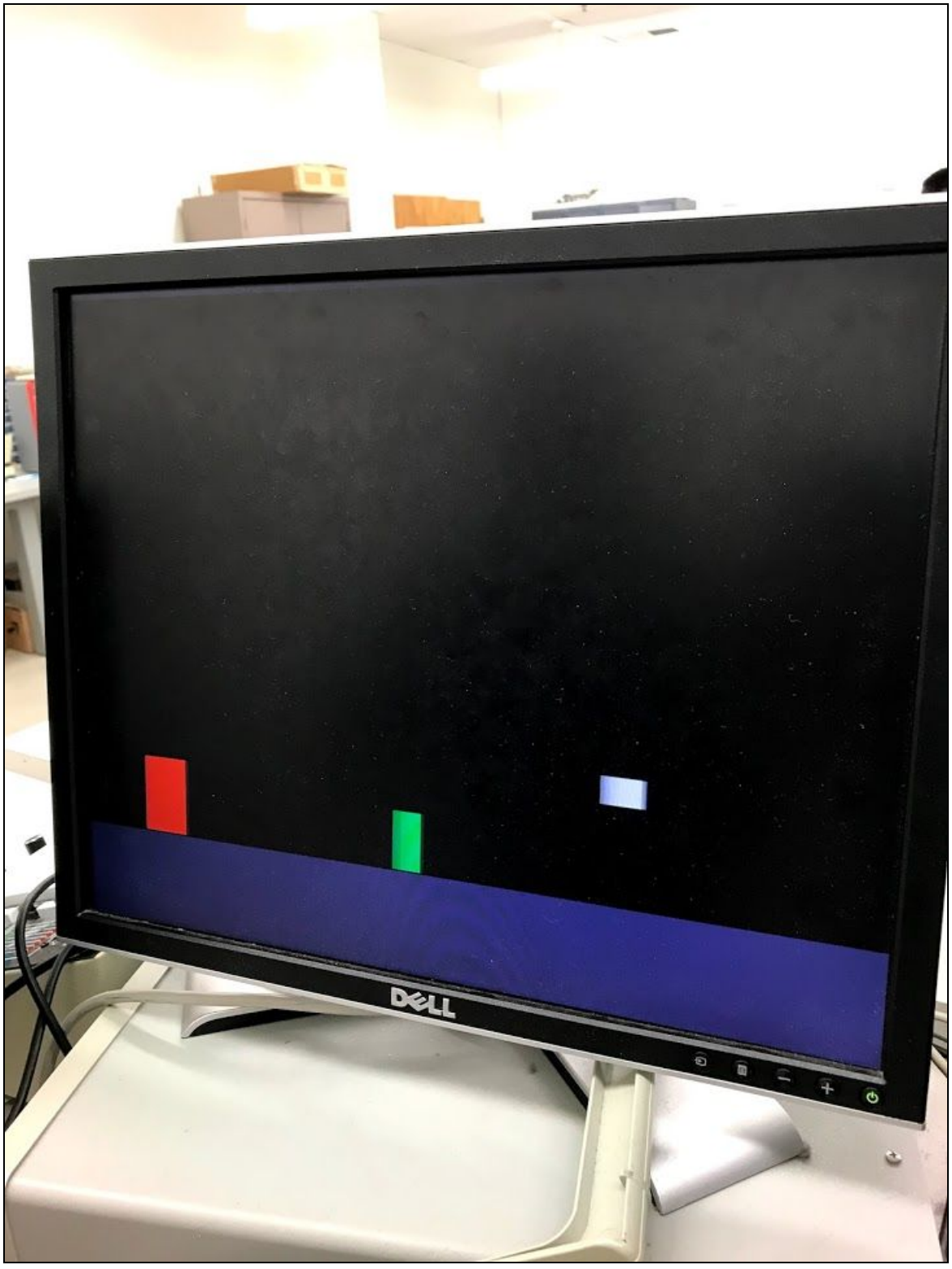

Figure 2: Screenshot of Google Chrome's T-Rex Game

At its basic design, a player will be playing as a scared dinosaur attempting to run away from an erupting volcano while faced with obstacles that are attempting to stop his escape. Through the use of the Nexys3 FPGA's onboard buttons (as well as switches for secret functionalities, as explained later in the report), the user will be able to input basic movement commands (jump and duck) to dodge pseudo randomly-generated cacti obstacles that approach the dinosaur on a scrolling playfield.

Additionally, as the player continues to survive during one playthrough of the game, a score will be incremented and displayed on the Nexys3's onboard seven-segment display. Similar to what was done in Lab 3, specific referenced manuals were explored as a means of clarifying our methods of achieving several implementations of our Dinosaur Game.

When originally submitting the proposal for this project, we did not know the exact extent to which we could properly implement our vision for the dinosaur game. However, we ended up successfully achieving all of the goals and functionalities we aimed to complete, as well as having a bit of time to implement an additional couple of features and perform quality-of-life upgrades to the game. Therefore, the rubric that we initially laid out for this project remains almost exactly the same, and is presented for convenience in the following paragraphs:

- Group Grading Rubric (from Project Proposal)
    - Player Input Handling (20%) - Based on the player's inputs, the dinosaur should behave accordingly (jumping and ducking), ensuring that edge cases are covered (no double-jumps, etc.).
    - Collision Detection (20%) - Once the player collides with an obstacle (cacti or pterodactyls), the game should promptly end, and the score should stop incrementing. The player should be made aware of the "Game Over" state (this will be defined later in implementation, potentially by a blinking score display, or an LED indication on the FPGA, or a "GAME OVER" message on the external display).
    - VGA Display (30%) - A visual representation of the entire game should be displayed through the use of the VGA cable on an external monitor. Simple, yet representative, sprites should be present for each of the objects (player, cacti, pterodactyls, ground, etc.).
    - Scoring Display (20%) - Based on the player's current game, a display of the player's score (either distance travelled or total time alive) should be present during the entire game on the FPGA's 7-segment display.
    - Reset Functionality (10%) - Upon indication by the player, a game can be reset to a "like-new" state, where the score, player position, and playfield is reset. There should be no indication of any previous run (hence, "like-new").

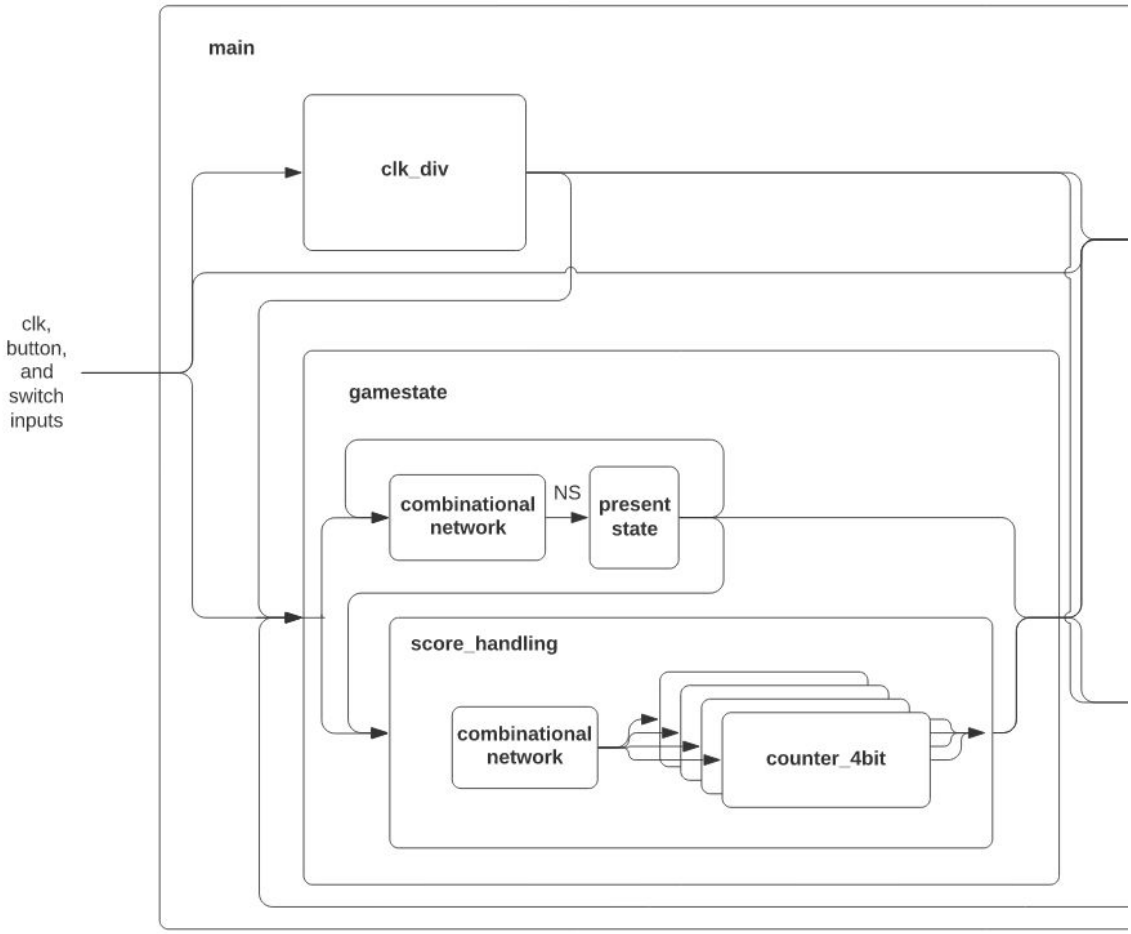

Figure 3: High-Level Block Diagram of Modular Structure

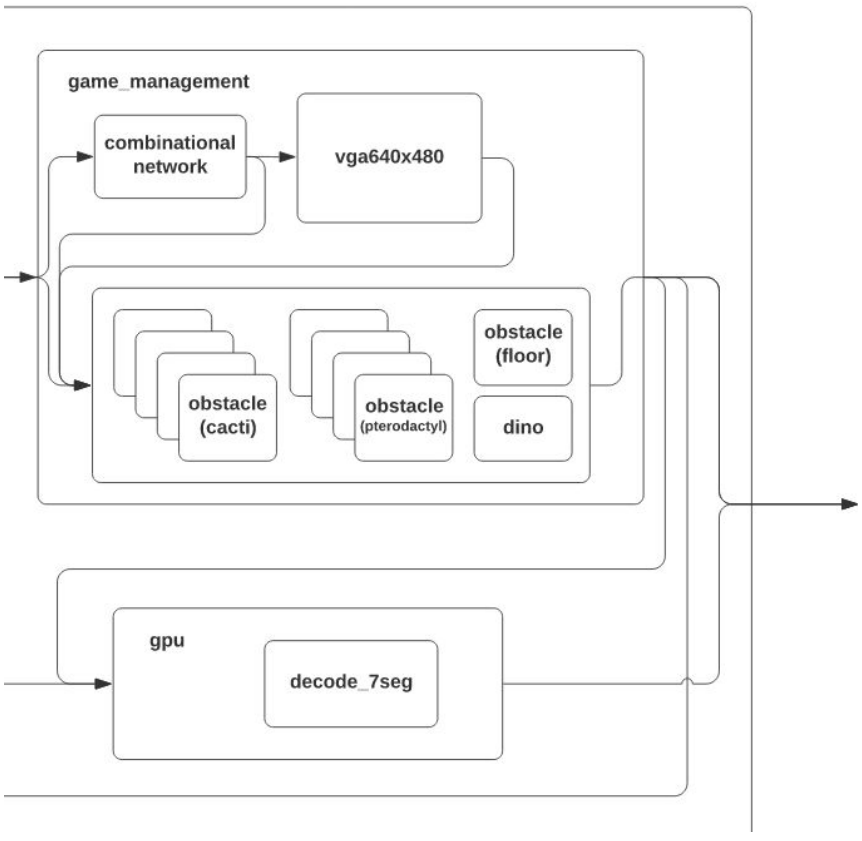

Figure 4: High-Level Block Diagram of Modular Structure (cont.)

As portrayed in Figures 3 and 4, a more specific breakdown of the design is given in the form of a high-level block diagram.

At the top level, we have the “main” module, which essentially provides housing for and connections between the multiple submodules that actually implement the required functionalities. Figures 3 and 4 shows these modules, describing the connections each of these submodules have with one another. In its simplest form, the input received from the user in the form of a button press or flipping of a switch is passed directly into the “gamestate” and “game_management” modules, which cause the game to behave accordingly based on the user inputs. These behaviors—and reasons for passing to the certain modules—are expanded upon in the following subsection of the report.

We proceed to explain each of these individual modules and submodules in greater detail:

I. clk_div:

The “clk_div” module achieves the implementation of a clock divider of the master 100MHz clock on the Nexys3 FPGA into 3 additional smaller-frequency clocks that are used by the other submodules of the parent “main” module. These modules include the “gpu” and “gamestate” modules.

Input: As input, the master clock line is passed in, as well as a rst line for means of resetting all of the counters used in the divider’s implementation to 0.

Output: As output, 4 lines representing each of the needed clocks are driven by the module, including a “display” clock used for refreshing the 4 digits on the 7-segment digit display, a “blink” clock used for determining the rate of the display’s blinking when the game is in a “GAME OVER” state, and two score clocks with different frequencies as a means of incrementing the player’s current score depending on whether or not the “Super Secret Switch 2” is set (explained in more detail with the “gamestate” module).

Method: The “clk_div” module utilizes a closely-similar method as that employed in previous projects. Essentially, for each output clock that we want to “divide up” our master clock into, we maintain a counter that essentially increments in value upon the posedge of every master clock cycle. Once the counter reaches some predetermined max value based on its intended frequency, the respective output clock line is toggled, and the counter is reset to 0. This process is repeated for the duration of the master clock’s operation.

II. gamestate:

The “gamestate” module achieves the implementation of managing the game’s state based on the different user inputs and events (collision) that occur during the game’s play. Essentially, since our game is built around this finite-state machine, the “gamestate” module is our “main control driver,” influencing how the system behaves at every point in time. Additionally, this module also manages the current player score, instantiating a “score_handling” submodule to specifically deal with this.

Input: As input, in addition to a master clock and reset line, a 1-bit input representing the detection of a collision event by the sister “game_management” module is taken in, as well as two 1-bit lines representing score clocks that determine how fast the player’s score should be incremented. Additionally, a 1-bit input representing the state of a user input switch (what we call “Super Secret Switch 2”) is taken in, which fundamentally determines which score clock to use (the slower or faster one).

Output: As output, a 16-bit bus containing the 4-bit binary representation of each of the 4 digits of the score record is outputted, as well as the current state of the game (represented by 2 bits, as there are 3 possible states for our game to be in).

Method: Based on the value of the module’s inputs and the present state, the next state is determined on the posedge of the master clock. Specifically for transition from the Grace state to the Play state, an internal “counter” variable is held, incrementing whenever a posedge of the master clock is detected while in the grace state. Whenever this counter variable reaches some max value, the counter is reset back to 0, and the state is moved into the Play state. Essentially, a pseudo-clock divider implementation was used to manage this transition. To manage the current player score, this module passes it off to the instantiated “score_handling” submodule.

III. score_handling:

The “score_handling” module simply manages the 4 4-bit counters that hold the current score of the player’s game, housing instantiations of these counters and managing the connections between each of them.

Input: As input, in addition to the master clock and a reset line, the module takes in a 1-bit score clock line that determines the rate of incrementation for the player’s score, as well as a 2-bit value representing the state of the game (outputted from the “gamestate” module), which helps determine whether or not the score should be incrementing at all.

Output: The only output for the “score_handling” module is a 16-bit bus that represents the values of each of the 4 digits used by the stopwatch for its display (each of the digits is represented with 4-bits). This output is used by the “gpu” module to properly display the player’s current score.

Method: As depicted in the high-level block diagram, the “score_handling” module instantiates 4 4-bit counter submodules as a means of recording the current score of the player during this playthrough of the game. The module itself simply manages and drives

the input lines that are passed into these counter submodules based on not only the inputs to the "score_handling" module, but also the carry outputs of the counter modules themselves (which serve as an indication of the following counter module to increment).

IV. counter_4bit:

The "counter_4bit" module simply achieves the implementation of actually holding the current value that is to be displayed for a certain digit of the score display.

Input: As input, the counter module takes in a clk and reset line, as well as a 1-bit line driving incrementation. Additionally, a 4-bit input representing the max value that the digit can hold is passed in.

Output: As output, a 4-bit output represents the currently-held value, as well as a 1-bit output representing a carry from overflow of the counter over its max value (specified as an input for the next instantiated "counter_4bit" sister module in the parent "score_handling" module).

Method: Based on its inputs determined by its parent "score_handling" module, the counter submodule will only increment its current maintained value when its increment input line is set to HIGH. Once the counter increments past its max value, it will reset its value to 0, setting the 1-bit carry output which is then used to drive the increment input line for the following digit. The least-significant digit is driven by the used score clock, determined by the parent "score_handling" module.

V. game_management:

The "game_management" module achieves the implementation of collecting all the game data required to display the correct pixels on our VGA display since vga640x480 will do the actual "drawing". The main set of game data required includes the positions of the cacti, pterodactyls, floor, and dinosaur. The data is ultimately used to create all the objects we need. It also manages the logic for collision.

Input: As input, the game management module takes in the board clock, 2-bit game state, and three buttons: reset, jump, duck. There's an additional "super secret switch" input as well.

Output: As output, there's a horizontal and vertical sync along with the red (3-bit), green (3-bit), and blue (2-bit) outputs needed for VGA display. Last output is a collision state.

Method: The game management module instantiates all the game objects and these objects are driven by the game state. Simply put, if we're in the dead state, the game is essentially paused. The mechanism behind how game management decides what the current pixel color is can be decomposed into three steps. First, it'll check if the current pixel being processed is inside any of the objects. If it is, record down for all objects whether they're in or out the current pixel. Next, check what object it was that the current pixel is in and assign into a 7-bit register the corresponding (R,G,B) value. Lastly, the 7-bit register is partitioned and packaged into the red, blue, and green wires for the

vga640x480 module to draw. It should also be noted that before the last step, the data that was collected can be used to check if the dinosaur has collided with any of the obstacles.

VI. vga640x480:

The "vga640x480" module achieves the implementation of managing the animation and drawing of the pixels onto an external display connected through the VGA port on-board the Nexys3 FPGA board. This module was developed by Will Green and is licensed under the MIT License (reference https://timetoexplore.net/blog/arty-fpga-vga-verilog-01 for more details).

Input: As input, this module takes in a base clock and a pixel clock strobe. Also, there's a reset input to restart the frame.

Output: As output, this module outputs multiple signals in order to create a working VGA display. The first two are the horizontal and vertical syncs. Then, there's a signal to indicate the blanking interval along with a signal to detect whether a pixel is actively being drawn. The next two are signals that indicate the end of the screen and also the end of an active drawing. Finally, there are two signals that essentially keep track of the current x and y position.

Method: Given that we borrowed this module from Will Green's FPGA Blog, we treated vga640x480 as a blackbox that drew pixels on the screen for our game.

VII. obstacle:

The "obstacle" module implements the cactus and the pterodactyls that the player has to dodge. The obstacle comes in two varieties, cactus, and pterodactyls. Each obstacle is initialized with an initial height and width, and for pterodactyls an initial height

Input: As input, this module takes in the jump and duck buttons, as well as the reset button. Additionally it takes in the main clock, the animation and pixel strobe clocks.

Output: The "obstacle" module is derived from the example "square" module, so it outputs its left and right, upper and lower, x and y boundaries.

Method: The "obstacle" module's main objective is to increment its position to travel left across the screen. Once it is off the edge of the screen, it sets a wait timer to delay when it next appears on screen. It also picks new random parameters. The pterodactyls randomize their y-value, while the cactus randomize their object height. Then for every frame during the waiting period, the wait timer counts down by a random number between 0 and 30 (this random number is determined as an output of an instantiated submodule "randomValue", which is explained in further detail in the next subsection), until it is less than 30. Then it appears on screen again and resumes traveling to the left.

VIII. randomValue:

The "randomValue" module achieves the implementation of producing a

pseudo-randomly generated value for the use of randomizing the wait-time for an obstacle after it travels "off the end of the screen," as well as randomizing the height value for cacti obstacle instantiations and the vertical y-position of the pterodactyl instantiations.

Input: As input, this module takes in a clock, reset, and seed.

Output: As output, this module produces a pseudo-random 4-bit value and the random 1-bit parsed from the value.

Method: The randomization is driven by a linear-feedback shift register. Essentially, it'll take the initial seed and shift it 1-bit to the left and replace the LSB with a random bit. This new value is the output. This "random bit" is generated by performing an XOR on the leftmost two bits of the previous output.

IX. dinosaur:

The "dinosaur"module implements the main character of our game. It is the object with which the player can control and interact to play the game. The dinosaur has 2 actions, jump and duck, with respective buttons to control them.

Input: As input, this module takes in the jump and duck buttons, as well as the reset button. Additionally it takes in the main clock, the animation and pixel strobe clocks.

Output: The "dinosaur" module is also derived from the example "square" module, so it outputs its left and right, upper and lower, x and y boundaries.

Method: Every animation cycle, the dinosaur first starts off by checking if it's in a jump phase. If it is it will update the position and velocity with simple kinematics, by incrementing the velocity by gravity (and optionally fast fall), and updating the position by the velocity. The jump phase ends when the Dino's position falls below the floor, where it is then placed back on the floor and velocity is reset to 0. If It is not in a jump phase, it checks to see if the jump button is being pressed, to start off the jump phase. If it is neither in jump phase, nor is the jump button being pressed, then the dino will check for the duck button. If it is pressed the dino will set its current height to the duck height value, and adjust its y value so that it does not lift off the floor (since the y value is at the center, lowering the height causes the top to move down and the bottom to move up). Finally, if the duck button is not being pressed, the height of the dino is reset to normal, and the y position is adjusted accordingly.

X. gpu:

The "gpu" module achieves the implementation of displaying the player's current score, as well as providing the implementation for blinking the display when the game reports the player's death ("GAME OVER" state).

Input: As input, a 2-bit input is taken in representing the current game state, the display clock and blink clock lines. Additionally, a 16-bit bus representing the current numerical values of the stopwatch is taken in.

Output: As output, the modules provides a 7-bit and 4-bit output representing the values to set the cathodes and anodes to on the corresponding digit of the 7-segment display, respectively.

Method: The basic idea behind the display is essentially that based on the posedge of the display clock (which has a high enough frequency to ensure that the human eye cannot notice the change in power associated with each segment in the 7-segment display), the value displayed is refreshed, looping through each of the 4 digits, and setting their respective cathodes and anodes to the correct values (determined by the outputs of the "gamestate" sister module). During the "GAME OVER" (or "player death") state, the entire display blinks at the frequency determined by the inputted blink clock. This is done by essentially "disabling" the setting of any of the segments of the display. The cathodes and anodes setting is essentially being overwritten during this period of time that the digit should be turned off.

XI. decode_7seg:

The "decode_7seg" module implements the actual translation of a 4-bit value to the proper setting for the 7 cathodes of the 7-segment display.

Input: As input, a 4-bit input representing the value to be decoded is taken in, as well as a 1-bit input representing whether or not the digit should be currently disabled (for blinking implementation).

Output: As output, a 7-bit output representing the proper setting of the cathodes for a 7-segment digit is outputted.

Method: The method is quite simple: a case statement is used to directly match every possible inputted value with its corresponding cathode settings. Before this case statement, however, if the disable input is set, then the cathodes are automatically set to all 0, no matter the state of the value input.

Simulation Documentation

The majority of our testing was done directly by fully-programming the Nexys3 FPGA board since that was the most practical way to see if our game features translated correctly into the graphical display. However, we did run some simulations to make sure our game state transitions were accurate under-the-hood:

We first tested to see if our grace period held for a certain amount of time. Once this grace period is over, it should immediately transition to the play state (as depicted in Figure 5).

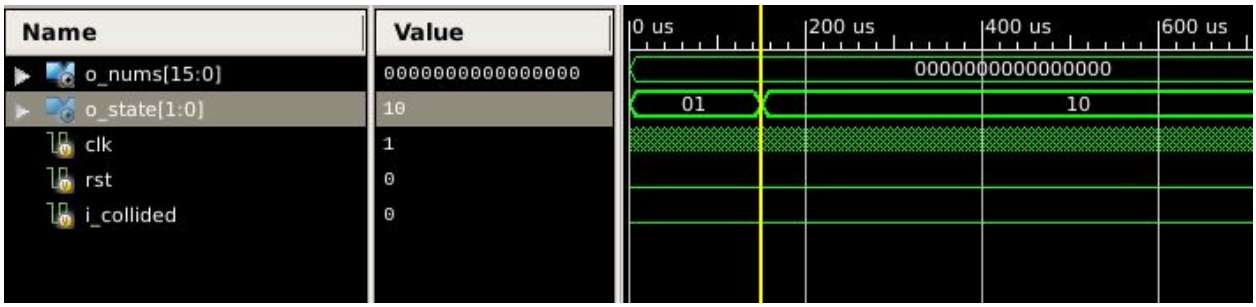

Figure 5: Transition from Grace to Play State

Next, we had to make sure that the score clock ran correctly during the play state (as depicted in Figure 6).

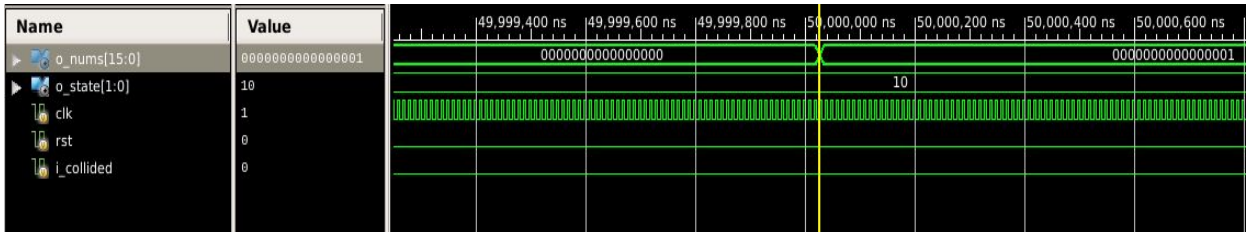

Figure 6: Score Clock Running During Play State

The third test case was to check for collision. Once the dinosaur collides with an obstacle, the game state should leave the play state and enter the dead state (as depicted in Figure 7).

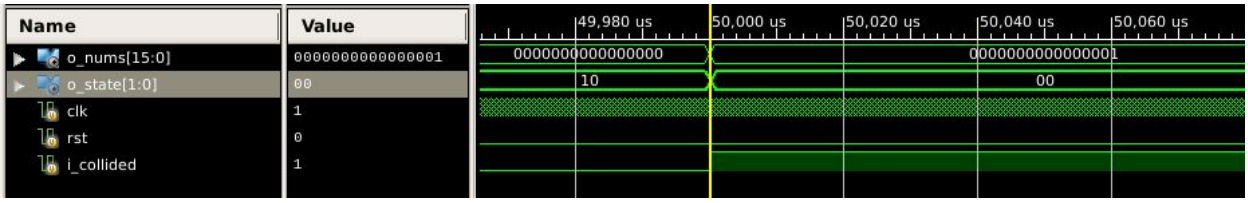

Figure 7: Collision Triggering State Transition from Play to Dead

Lastly, the game state should recognize when the player resets. Whether they're in the play state or they're in the dead state, reset should force the game state into grace period (as depicted in Figure 8).

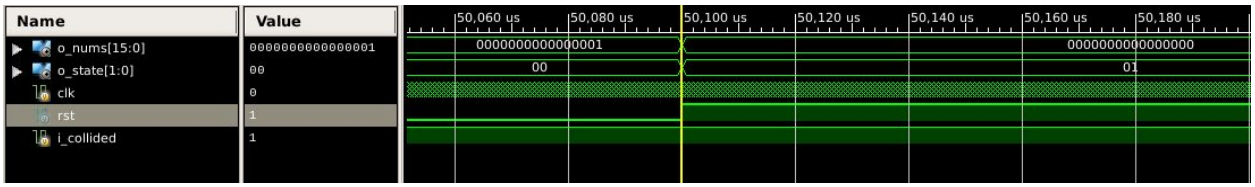

Figure 8: Reset Triggering a State Transition from Dead to Grace

Besides just playing the game, there were three critical test cases that should be mentioned. First, we had to check if gravity worked and that jump caused the dinosaur to move to an appropriate height. In addition, fast-fall should allow us to move to the floor faster. Figure 9 depicts each of these simple tests of functionality.

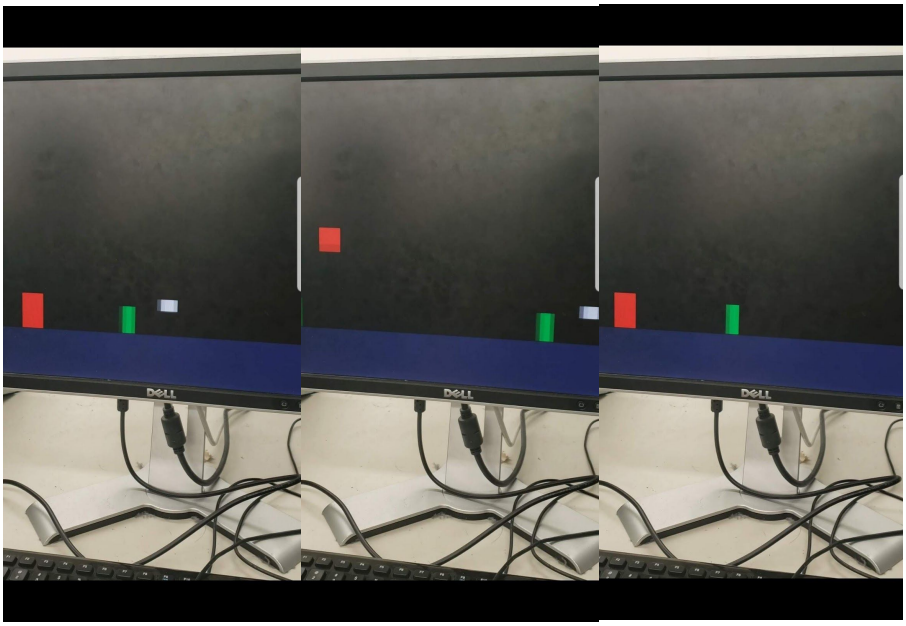

Figure 9: Depiction of Implemented Jump
and Fast-Fall Feature (Ducking While Jumping)

The second test case was to make sure our randomizer worked. The heights of the cacti, y-positions of the pterodactyls (as Figure 10 depicts), and the time of entrance of obstacles should all be randomized.

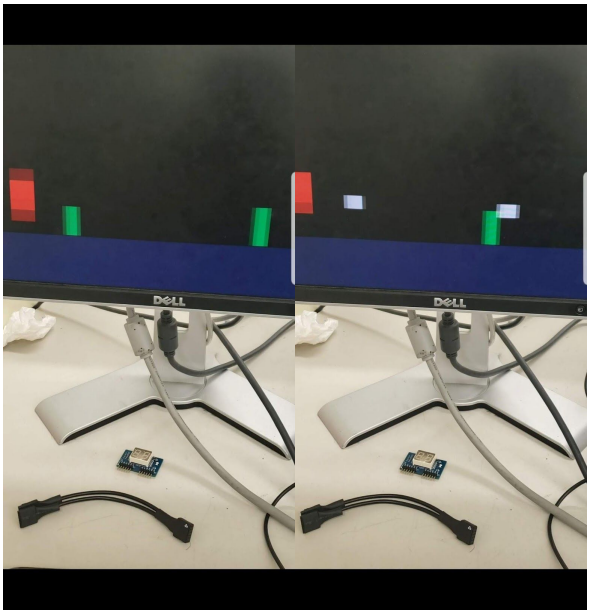

Figure 10: Randomized Obstacles

The third case was to ensure that the score clock display was working (depicted in Figure 11). We also tested for blinking when the player dies, but that's difficult to portray through static images.

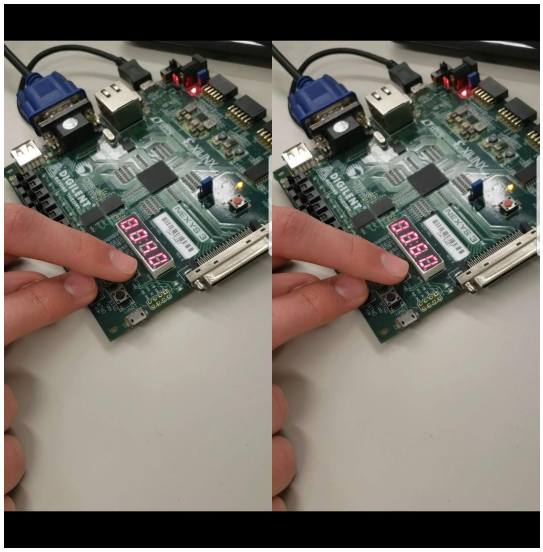

Figure 11: Score Clock Counting

Throughout the two weeks of developing this game, we experienced several major bugs. The most prominent one being that the dinosaur would fall through the floor when fastfall was used during a jump. This was difficult to debug because our logic for floor collision check was correct. We make sure that if the dinosaur's midpoint is greater than its normal standing value in the y-axis, then position the dinosaur back to that normal value. Clearly, the collision logic wasn't the error. So, we deduced that the logic itself wasn't being called. After tracing through the dinosaur logic code, we discovered that since the duck is inputted during the jumping state, the midpoint condition was never being tested. Thus, the dinosaur wouldn't get out the floor until a normal jump was inputted to restore it back to its normal position. A simple conditional to check if the dinosaur was ducking during a jump was added and setting its current height to its appropriate value fixed this issue.

The next two sets of bugs involved the cactus and pterodactyl obstacles. During the initial segment of the game after grace period, both the cacti and pterodactyls would enter from the left at the same time. The confusion was that after their initial passings, they'd enter at distinct intervals so that no two obstacles entered simultaneously which we wanted. So, there had to be an issue in one of the initialization processes. It turned out that even though we were ensuring each instantiation of obstacles has a different wait time, we forgot to make a distinction in terms of wait time between the pterodactyl and cactus. Our solution was to add a half factor to the wait times for pterodactyls to create a big enough time gap between the pterodactyls and cacti. The second obstacle bug involved the positioning of pixels. Parts of a single cactus and pterodactyl would be apparent on the right edge of the screen. As more obstacles came in, the parts would sometimes disappear as well. This turned out to be just a dimensional error when we set our initial x positions in our game management module. A simple offset by 20 pixels to the right moved them off the screen and prevented any parts from showing up throughout the game.

Our final set of bugs dealt with our score clock. The first bug was that our score clock wouldn't run in simulation. Despite the clear game state transition from grace to play, the score clock would still be frozen at zero. All the carry over and counting logic worked because it was essentially derived from our stopwatch module (done in Lab 3) which worked. The second bug was that during the grace period, the display for our score blinked when it shouldn't have. Blinking should only occur when the dinosaur has collided with an obstacle. For each of these bugs, we made the same error - forgetting to set the correct amount of bits. Since we didn't specify two bits for the game state, the state passed in was truncated and thus score handling assumed the game was never in the play state. So, counting didn't occur. As for the blinking, we also didn't specify the bits for the game state. The reason why the display was blinking during grace was that the grace state is labeled (01) and the dead state is labeled (00). So, the truncation

resulted in a value of 0 making the gpu module think that the game is in the dead state and thus enable blinking.

Contribution

In terms of the other individual contributions that each of us had to the design and implementation of the Dinosaur Game project, our group worked closely together throughout every other step of the process. Even though each of our group members worked and focused on another portion of the project themselves, each of us worked through the problem-solving aspect and implementation design of each module as a means of not only ensuring the modules' validity but also ensuring that each of us worked equally towards the completion of the project.

As a means of being a little more specific, Harrison specifically focused on the management of the game's state along with Eldon, as well as building the entire organizational structure for the high-level design and graphical display. Over the course of its implementation, this design was further refined to incorporate and account for the various functionalities and bugs we discovered, respectively.

To dive a little deeper into the "gamestate" module that Harrison and Eldon focused on, as seen in Figure 5, there were fundamentally 3 possible states for our game to be in: Grace, Play, and Dead. As mentioned earlier in the "Problem Statement and Design Description" section, the "gamestate" module, based on the present state of the game and the value of its inputs (which include a clock and reset line, as well as a collision event line), would determine when to transition to a new game state. When the RESET line is held high, the next state is always the Grace state, as this is the first state we want to be in to allow the player to "become oriented" with the controls for a short amount of time (arbitrarily set to about 3 seconds) before being faced with incoming obstacles. As mentioned before, over the course of about 3 seconds, a counter is incremented up to a max value, triggering a state transition to the Play state when reaching that max value. When in the play state, the game operates normally, meaning that the

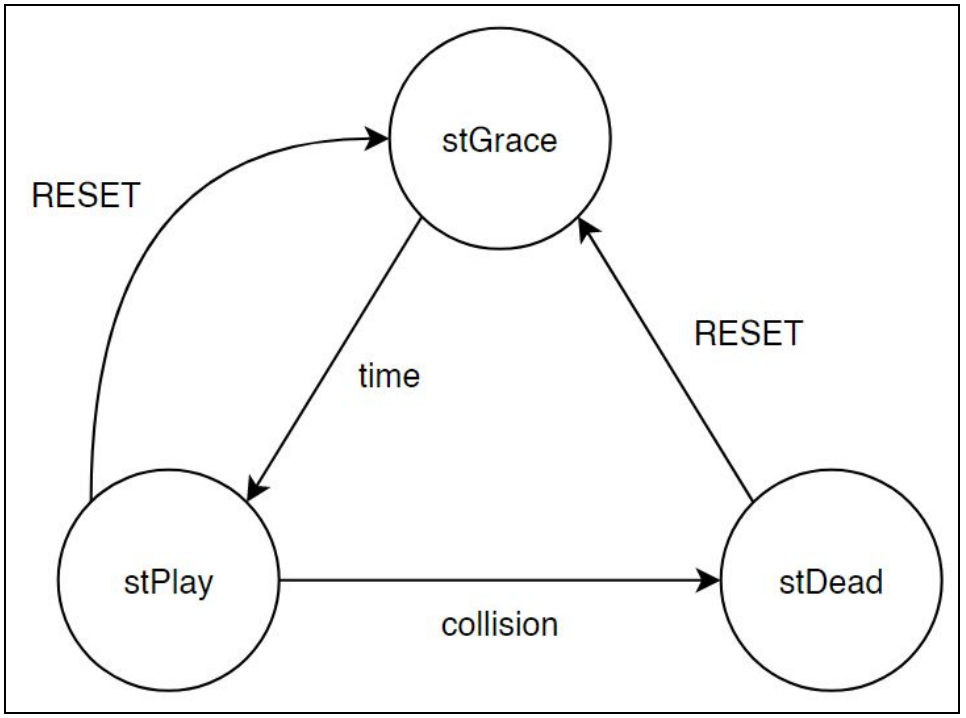

Figure 12: State Transition Diagram for our Dinosaur Game Implementation

obstacles begin pseudo-randomly generating and approaching the player.

Eldon specifically focused on the “randomValue” randomizer module used to produce randomized values for use by the instantiated obstacles. Again, this was done with a linear-feedback shift register which takes advantage of bit-shifting and XOR-ing certain bits. This allowed for the randomization of the heights of the cacti, y-positions of the pterodactyls, and the wait time delays. In addition, Eldon calibrated certain parameters such as the obstacle velocities along with how high the cacti (size-wise) and pterodactyls (position-wise) should be. The purpose of this was that the game would be unfair and a pain to play otherwise. The obstacles would come in too fast or too slow. An excessively tall cactus would result in instant death while a high-positioned pterodactyl would defeat the purpose of ducking. Lastly, Eldon commented each module to make sure that if anyone wanted to add additional features, they’d easily be able to understand the base code.

Jasper specifically focused on implementing the “gpu” and “deocode_7seg” modules, as well as incorporating collision detection into the “game_management” module, which serves to drive the collision detection event line input into the sister “gamestate” module.

The “gpu” module, which actually implements the method of displaying the player’s current score on the Nexys3 FPGA’s on-board 7-segment display, was adapted from our same custom implementation produced in the Lab 3 Stopwatch project completed in the previous weeks of the class. The most notable change was the removal of functionality regarding “adjust mode,” as it was no longer needed. The “adjust mode” blinking functionality was repurposed to instead blink all of the digits when the game entered the Dead state. The “decode_7seg” module remained virtually unchanged.

The VGA module was borrowed from the “Time to Explore” blog by Will Green (reference https://timetoexplore.net/blog/arty-fpga-vga-verilog-01 for more information), and remained unchanged outside of the drivers for and uses of the inputs and outputs.

Additionally, Jasper implemented the logic used for collision detection within the “game_management” module. Since the implementation of the drawing already checked each pixel to see what objects it contained, and whether it was a pterodactyl, cactus, dino, or the floor, we already had boolean flags for each of those collisions. To detect if a collision happened that would result in a player’s death (collision between the dinosaur and either a cactus or pterodactyl), it was as simple as adding another flag that checked if the pixel was inside the dinosaur and either a cactus or a pterodactyl.

Jasper handled the initial writing of the dinosaur logic. In the “dinosaur” module, processing of the raw jump button input was included, performing a form of debouncing by fundamentally

ignoring any jump button presses during the time interval that the dinosaur was jumping (and since this jumping time interval was much greater than the fractional amount of time that a physical button “bounces” for, this processing proved successful in its intended behavior). Additionally the duck button did not need to be debounced, since it was designed to be held down or released to trigger events. Jasper handled the main logic of the dinosaur, which was the jump activation and physics. The other logic for the dinosaur to implement was the duck functionality. When the button was down the dino would duck and its height and position would be adjusted accordingly. If the button was not held, position and height were reset to normal.

Upon initial tests of the implementation of the “dinosaur” module, the dinosaur would tend to sometimes act incorrectly (for example, embedding itself deep into the ground and seemingly ignoring collision with the floor, as well as sometimes floating while ducking). Eldon and Harrison focused on the subsequent debugging and analysis of the logic implemented within the “dinosaur” module, making most notably 3 major changes. First, they fixed some pixel boundary issues, so all positional features would work smoothly. Secondly, the code was reorganized in that the begin and ends were moved and marked since an always block was ending too early, meaning that not all of the input cases were being addressed at the proper clock interval. Finally, they covered the ducking while jumping case which was causing the dinosaur to fall through the floor (all of these are explained in further detail in the debugging subsection of the “Simulation Documentation” section above).

As a means of attempting to cover all the possible edge cases, each of our members also worked specifically on testing each specific module/portion of the lab themselves. Although the simulation software presents a convenient way to debug, as it allows the developer to expose certain internal variables and values at any point in time, many of these tests were done on hardware, as it oftentimes presented itself as a faster alternative to identify which specific state-transitional behavior is not properly handled by the system. Compiling these tests done by each team member formed an extensive list of tests that we used to confirm the validity of the entire project in implementing these functionalities. Even though each of the group members focused on a specific module for testing, we all worked closely together to ensure that every possible case was covered (and for those that we did not plan for immediately, we discovered through our simulation and bug-testing efforts described in the “Simulation Documentation” section).

Conclusion

In conclusion, our final Dinosaur Game project, which fundamentally mimics Google Chrome's T-Rex No-Internet Connection game as closely as possible, was implemented with a top module "main" instantiating several submodules (including a clock divider "clk_div" module, main game state management "gamestate" module, main game logic "game_management" module, and a 7-segment display "gpu" driver module). This top module simply served as a means of encapsulating the instantiated submodules, providing nice interfaces utilized by the Nexys3 FPGA to provide input to the game and drive the 4-digit 7-segment display system, as well as providing the wired connections between each submodule's inputs and outputs. The Dinosaur Game's functionalities that are implemented by this top "main" module's submodules include the ability for a user to have the dinosaur player jump and duck, generate pseudo-random obstacles—in terms of size or vertical position, for the cacti and pterodactyls respectively—for the player to avoid, collision detection for the means of determining when the player would "lose", display the entire game on an external monitor connected with VGA, and display the current score for the game in progress.

In addition to these base functionalities, as mentioned before, our group had enough time to implement multiple additional functionalities, most notably a "fast fall" mechanic, allowing the user to fall to the ground faster by inputting duck while currently jumping. This allows the game to feel more "skill-based" while also bringing the game to more closely mimic Google Chrome's T-Rex version, as they have implemented this same mechanic. Additionally, for the intention of adding some more "fun" to the game, we as the developers and creators of the game decided to include two "super secret switches" that each added a heavy modification to the game. One of which allowed the dinosaur to jump almost twice as high, allowing the game to become a lot easier, as timing your jumps was not as big of an issue. The second switch we deemed our "cheat code switch", which causes our score to increment close to 10x as fast as normal, making it super easy for us to achieve the highest score that will ever be made on our game (we are the developers after all, so it's only fair).

One of the major difficulties we faced during the development of this project was simply getting started on the design. We knew our eventual goals for the Dinosaur Game project, however figuring out where to start developing the source code from scratch was something that we were having a hard time with. Attempting to understand how to break down the task into smaller portions and operations to be implemented by separate modules presented as a harsh difficulty towards the beginning of the project—and not having any code to work off of means that implementing each little functionality involved a decent amount of debugging of our produced Verilog source code. We overcame this difficulty by creating a high-level block diagram (seen in the first section of this report) and working directly with that throughout the whole process. This allowed us to compartmentalize each of the functionalities we needed to implement for our

Dinosaur Game project, as well as allowing us to understand the specific control flow and data flow through each module and submodule. Once we understood this, it allowed us to understand how each module interacted with each other, and gave us a strong foundation and starting points for implementing the required functionalities. Determining the interfaces for each module—as well as the interactions of each submodule with each other—was the next step, making the overall development process a lot easier to take on and complete.

Another major difficulty we faced was working with the Verilog language as well as the Xilinx ISE and ISim software. For each of us, this is the first time we worked on a full-scale project from complete scratch—no explicit external specifications or source code for the Dinosaur Game were given to us to start—with this language and software, and therefore we referenced numerous external documentation and tutorial sources in an effort to learn how to effectively code in and work with the design software. Several resources and lecture notes accessed from MIT's OpenCourseWare website were referenced as a means of clarifying not only specific syntax details but also higher-level concepts that Verilog and the Xilinx ISE software can achieve. On a more specific note, the Nexys3 Reference Manual provided to us on the specification document was particularly helpful in understanding how to work with the onboard 4-digit 7-segment display (specifically how to control it). Additionally, the multiple resources provided on the CCLE course website for Verilog were extremely helpful during the development of this project by helping aid the translation of our design ideas discussed earlier into the Verilog code (and therefore also translating the design into actual hardware). Multiple references were also made to the previous labs as well, allowing us to recognize how to achieve certain simple operations and designs in Verilog.