# Dino Run

Group Members: Ryan Lo, Nicole Wong, Ramya Satish

Lab Section: 3

Demo Date: 12/8/17

TA: Hongxiang Gu

**Introduction**

For our final project, we designed and implemented Google Chrome's endless runner game, unofficially known as the *T-Rex Offline Game.* On Google Chrome, this game is found when the user's computer is offline, and can be played while the user waits for internet connection.

The player is presented with a 2D world with a desert ground. The main character is a red T-Rex that stands on the ground, to the left of the world. The dinosaur then runs on the ground to the right, but the background is moving toward the left. Then, at random intervals, a randomly selected hazard is placed at the right end of the screen and starts moving towards the dinosaur: either a pterodactyl or a cactus. If either touch the dinosaur, the T-Rex dies and the game ends. The cactus is placed on the ground, but the pterodactyl is randomly either on the ground or slightly above ground, thus requiring differently timed jumps to avoid. The goal is to keep the T-Rex alive for as long as possible, with the player's score being a multiple of the time alive. An image of the game is shown in Figure 1.

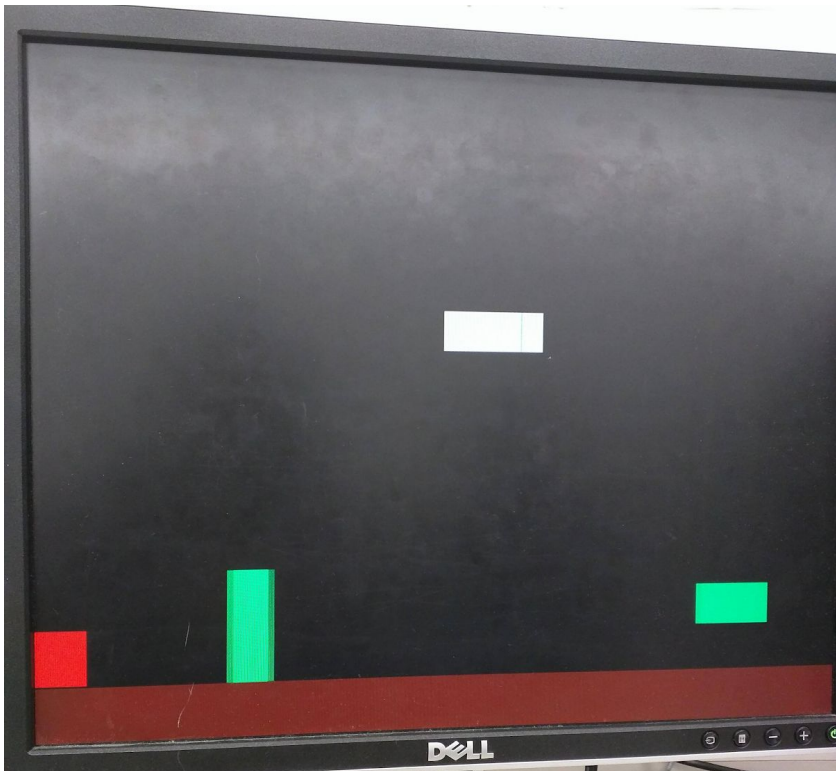


*Figure 1: Photo of Dino Run Game.* This image shows the VGA display of the game. The dinosaur is the red square on the left of the screen. There are two obstacles on the right: a tall cactus, and a pterodactyl in the air. The scrolling background consists of the white cloud at the top of the screen and the obstacles, both moving leftward.

To control this game, we assigned one of the FPGA's built-in buttons for making the dinosaur jump. Every time the jump button is pushed when the dinosaur is on the ground, the T-Rex will jump vertically in a motion that makes it appear natural, as if the dinosaur is impacted by gravity. The player is not allowed to spam the jump button; in other words, the dinosaur will not remain in the air if the button is held or jump again in mid-air.

Additionally, another button resets the game. The score will revert back to 0 and the dinosaur will begin its run again without any obstacles in its way (until they are randomly generated). This button can be used in the middle of the game to reset the score, or after the user loses, if the user wishes to restart the game.

The seven-segment display will display the user's score in the current game. If the dinosaur is still alive, the score will increment at a frequency of about 3 Hz. After a score checkpoint is reached, the game is sped up such that the screen scrolls more quickly and obstacles appear more frequently. Once the dinosaur collides with an obstacle and dies, the score stops incrementing. The seven-segment display will blink with the final score, and the VGA display will be black, until the game is restarted by pressing the reset button.

The game was run using the Nexys 3 FPGA board and was displayed on a monitor using VGA output. The built-in buttons on the FPGA board were used to control the dinosaur and restart the game.

**Overall System Design**

The overall system design was achieved in Verilog through a top-level main module in a file called NERP_demo_top.v. This module took in a 1-bit master clock input (*clk*), 1-bit reset button input signal (*clr*), and a 1-bit jump button input signal (*jump*). The outputs for the top-level module were an 8-bit segment encoding for the 7-segment display (*seg*), a 4-bit anode enable signal indicating the indices of the digits to display (*an*), a 3-bit red VGA output (*red*), a 3-bit green VGA output (*green*), a 2-bit blue VGA output (*blue*), a 1-bit horizontal sync output for display (*hsync*), and a 1-bit vertical sync output for display (*vsync*). Figure 2 below portrays the design.

First, the reset and jump buttons are passed through the debouncer module to filter out noise and stabilize input. The output of these modules are the reset and jump states, which will serve as input for other modules in Figure 2. The clock divider module takes in the master clock and outputs clock signals which each have different purposes, ranging from VGA display to seven-segment display to graphics movement in the game.

The counter module increments the score at a frequency of about 3 Hz, and stops incrementing when the user is no longer alive. The seven-segment display module displays the score and outputs the seven-segment display fields at either a regular display frequency (if the user is alive) or a blinking frequency (if the user is dead).

The obstacle picker and the timing chooser both take in a clock signal, a reset signal, and an enable signal, and both output a random 8-bit number used in random obstacle generation via the LFSR approach.[1] The cloud module takes a clock signal and a reset signal in order to output the coordinates and the velocity of the moving cloud in the screen background. The obstacle module uses a clock signal, a reset signal, and two randomly generated numbers to randomly select a type of obstacle and randomly time the obstacle placement. The enemy module uses a clock signal, an obstacle signal, and the other obstacle's position to randomly select a type of obstacle and randomly time the placement such that it is not too close to the

other obstacle (so that it is possible for a player to clear both obstacles). The dino fields module takes the jump signal, the reset signal, and the dino_clk signal to determine the dinosaur's coordinates and velocity.

Finally, the VGA display takes the dinosaur coordinates, the enemy coordinates and dimensions, the obstacle coordinates and dimensions, the cloud coordinates, and the display clock to display all of this on the monitor using red, blue, green, hsync, and vsync signals.

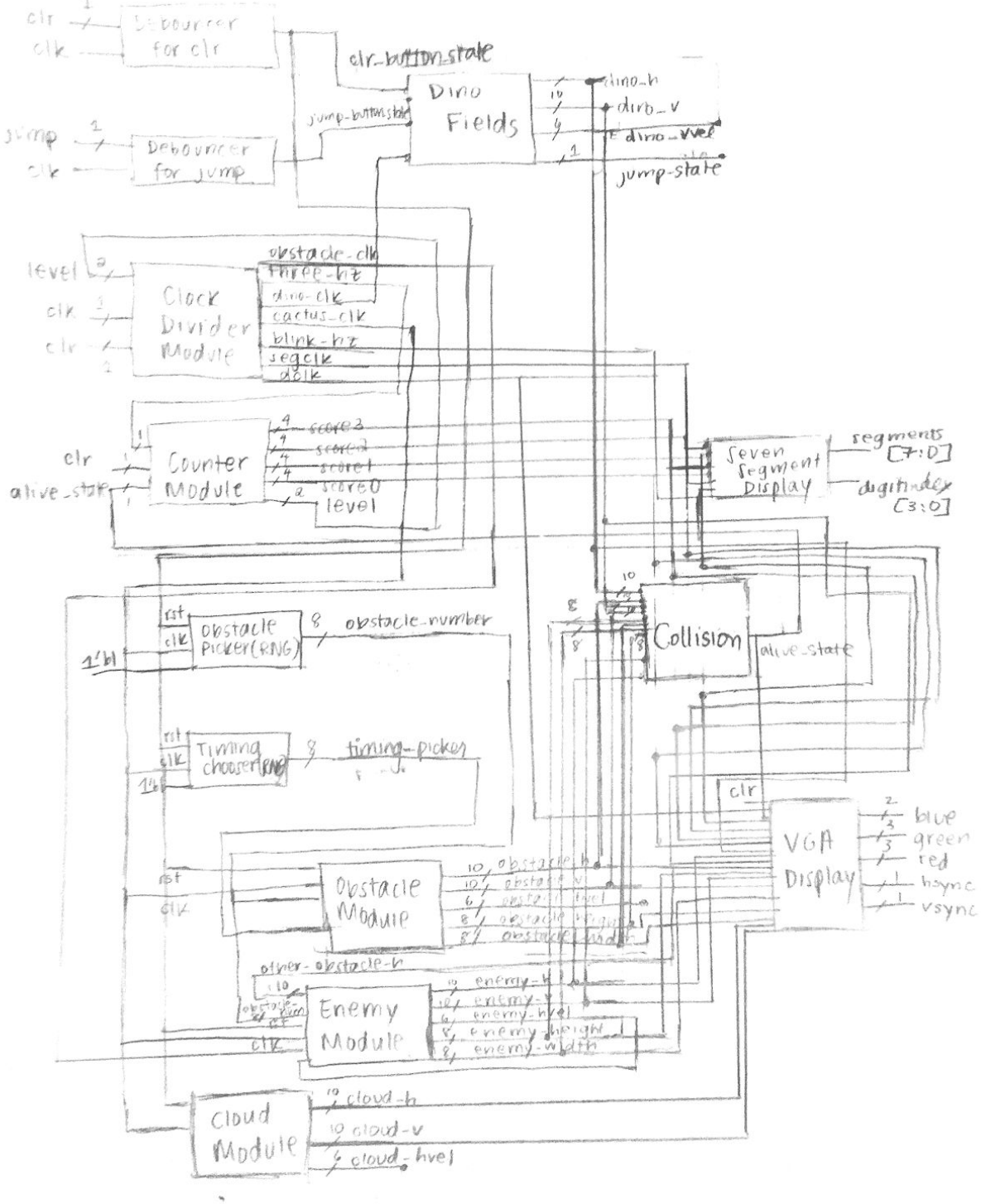
clr
clk
Debouncer for clr
clr_button_state
jump
clk
Debouncer for jump
jump_button_state
Dino Fields
dino_h
dino_v
dino_vvel
jump_state
level
clk
clr
Clock Divider Module
obstacle_clk
three_hz
dino_clk
cactus_clk
blink_hz
segclk
dblk
clr
alive_state
Counter Module
score3
score2
score1
score0
level
Seven Segment Display
segments [7:0]
digit index [3:0]
rst
clk
obstacle Picker(RNG)
obstacle_number
1'b1
Collision
alive_state
rst
clk
Timing chooser(RNG)
timing_picker
1'b1
clr
VGA Display
blue
green
red
hsync
vsync
rst
clk
Obstacle Module
obstacle_h
other_obstacle_h
Enemy Module
enemy_h
enemy_v
enemy_hvel
enemy_height
enemy_width
clr
Cloud Module
cloud_h
cloud_v
cloud_hvel

*Figure 2: Overall System Design*. The top module takes in the master clock and button signals. It outputs the 8-bit segment encoding and 4-bit anode enable signal (*digit_index* in the diagram) for the seven-segment display, along with VGA output (*red*, *blue*, *green*, *hsync*, and *vsync*) for the monitor.

**Debouncer Module**

In order to remove noise and stabilize input upon pressing a button, the debouncer module was created. This module is analogous to the debouncer module in Lab 3, the stopwatch lab. The module is illustrated in Figure 3.

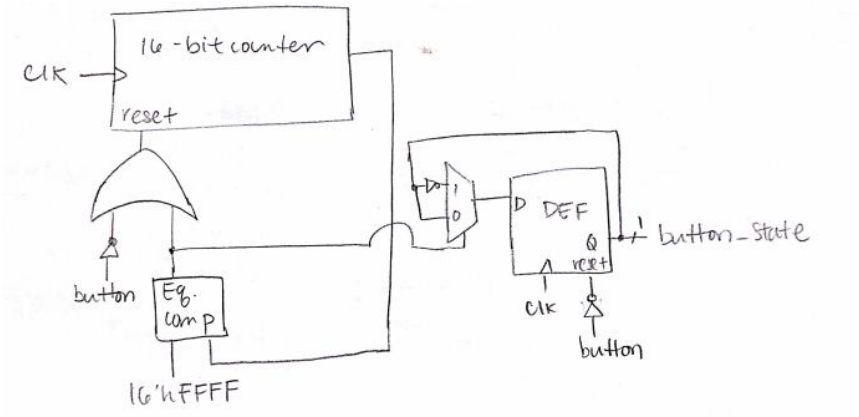


*Figure 3: Debouncer Module*. The debouncer module takes in the master clock and button signal, which are both 1-bit inputs. The output is the 1-bit final button state.

The debouncer stores the button state using a 1-bit DFF and 16-bit counter. If the button is not pressed, the counter and the final button state registers stay at 0. If the button is pressed, the counter increments by 1 every clock cycle. Once the counter reaches the maximum value (0xFFFF), the counter is reset and the button state is negated.

In the game, the debouncer module is used to generate the jump and reset states based on the jump and reset button presses.

**Clock Divider Module**

The game requires many different clock frequencies for a variety of purposes. The three hertz clock is used for the counter, dino_clk is used for the jumping dinosaur, cactus_clk and obstacle_clk are used for the obstacles, seg_clk is used for the seven-segment display, blink_clk is used for the blinking seven-segment display in the game over state, and dclk is used for the VGA display. Because the master clock (50 MHz) is too fast for each of these frequencies and, thus, would not display the game at a viewable speed, we designed a clock divider module to divide the master clock frequency to output the frequencies we wanted.

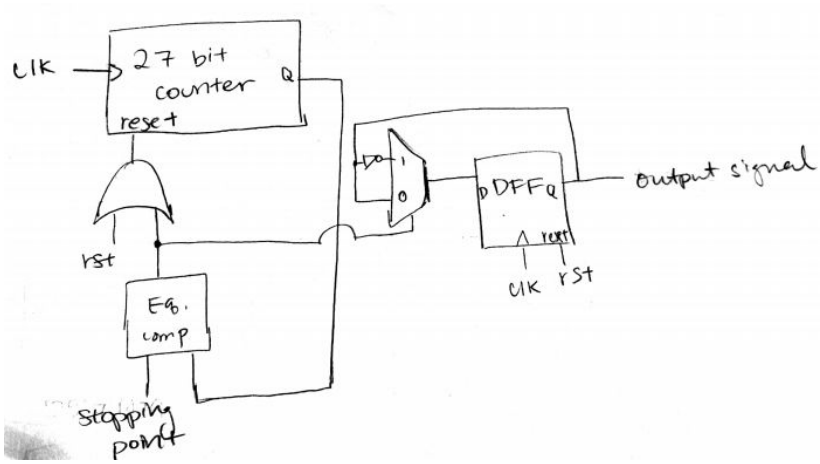


*Figure 4: Clock Divider Module.* The clock divider takes in the master clock as input, which is 1-bit, to generate the seven output frequencies. The seven output clock signals use identical logic to the above circuit, with the only difference being the counter stopping point for each frequency.

When the master clock signal reaches the positive edge, each of the counters increments its count by one. Each of the output clock signals has a "stopping point" for its counter to attain the desired clock frequency. The "stopping point" for the 3 Hz, dino_clk, cactus_clk, obstacle_clk, seg_clk, blink_clk, and dclk frequencies are 16777216 ($2^{24}$), 2097152 ($2^{21}$), 2097152 ($2^{21}$), 1073741824 ($2^{30}$), 131072 ($2^{17}$), 33554432 ($2^{25}$), and 2 ($2^{1}$) respectively. If the player has progressed far enough to move onto the next level, then the cactus clock becomes faster by only having to count up to 1048576 ($2^{20}$). Once the stopping point for any of the new clocks is reached, the register containing the bit-value of that clock is inverted and the counter for that clock will reset to zero.

Furthermore, one slight difference with the cactus_clk is that the counter is selected based on the input level. Once a certain score checkpoint is passed, the 2-bit input, *level*, is set to 1. The two values of *level* (0 and 1) cause us to pick between two values for the cactus_clk stopping point. This would essentially be a simple multiplexer checking whether the level is 0 or 1, and then selecting either 2097152 ($2^{21}$) or 1048576 ($2^{20}$), respectively, for the value of the counter.

**Counter Module**

The counter module is responsible for keeping track of the player's score. It takes as input the 3 Hz clock, the reset signal, and the alive state. The output of the module is the four digits of the score. More specifically, *score0* is the ones digit of the score, and *score3* is the thousands digit of the score. The

module also outputs a 2-bit *level* based on the score. The counter module is illustrated in Figure 5.

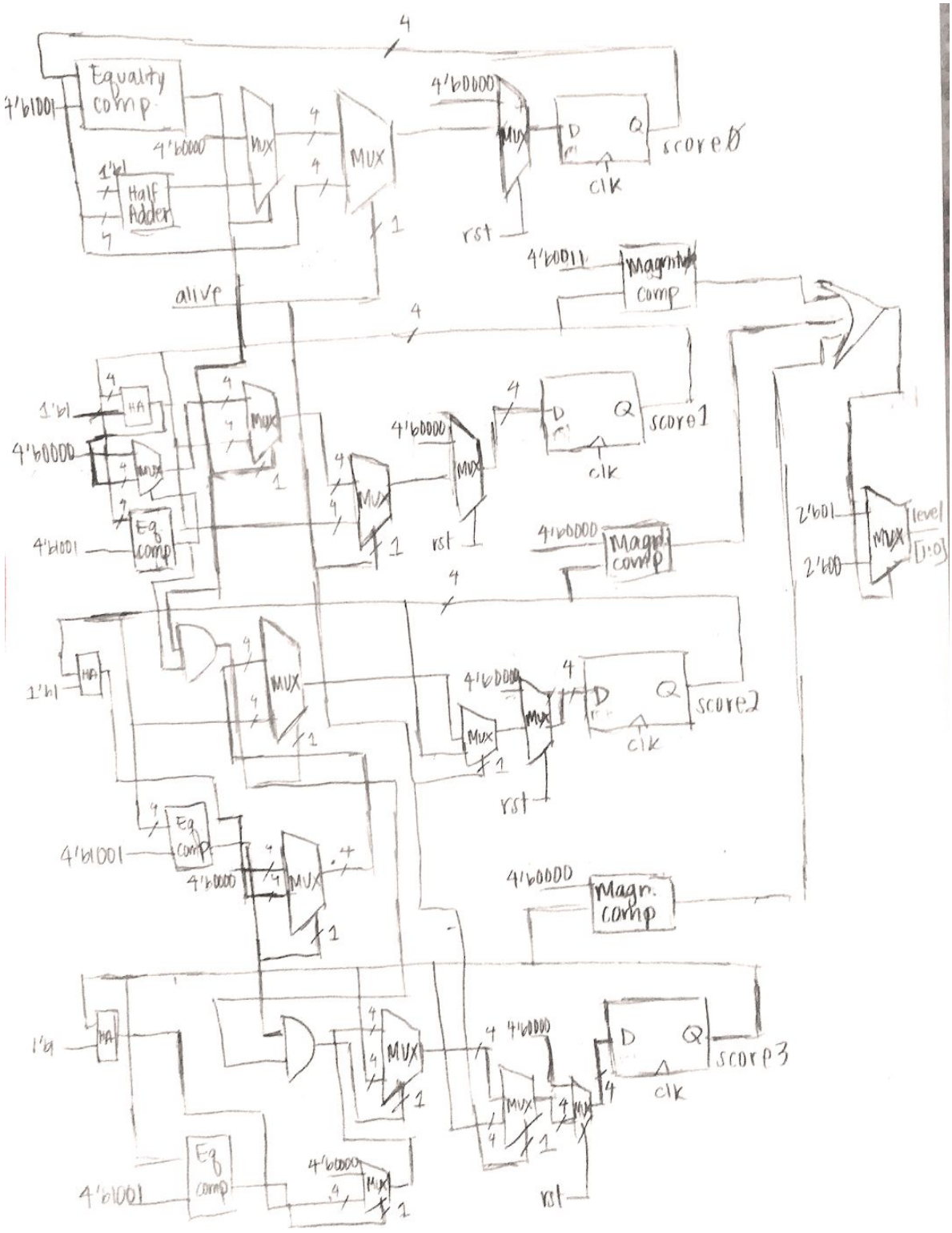


*Figure 5: Counter Module.* The counter module works by incrementing the four-digit score once every clock cycle (where the clock frequency is 3 Hz). The schematic consists of several multiplexers used to

check the carry-over of each digit upon incrementing the counter. If the user dies, the score digits are all the same as they were in the previous clock cycle. If the user hits reset, then each digit of the score is reset to 0. Finally, after a score checkpoint, the level is set to 1. If the score is below this checkpoint, then the level is 0.

At every clock cycle of the 3 Hz clock, the score0 (one's digit) is incremented. If the one's digit is a 9, then it is set to 0, and the ten's digit (score1) is incremented. If the ten's digit is *also* a 9, then the ten's digit is set to 0 and the hundred's digit (score2) is incremented. If the hundred's digit is *also* a 9, then the thousand's digit (score3) is incremented.

If the score is larger than a certain checkpoint (selected to be 40, as this is a reasonable point based on the difficulty of the game), then the level is set to be 1. If the score is below this checkpoint, the level is set to be 0. We checked that the score was greater than or equal to 40 by ensuring that *score1* was greater than 3, or *score2* or *score3* were larger than 0.

The level was later used as the input in Figure 4 when selecting the frequency of the cactus_clk for the obstacles. Furthermore, the four score digits were used as inputs for the seven segment display module in Figure 7.

**Seven Segment Display Module**

This module takes in the four 4-bit numbers from the counter module representing the four digits of the player's score, the alive signal that indicates whether the dinosaur is still alive, and the display and blinking clock frequencies (segment_hz and blink_hz respectively) from the clock divider module. The output of the module is an 8-bit segment encoding along with a 4-bit digit index to display the correct numbers on the FPGA board, displaying the player's score.

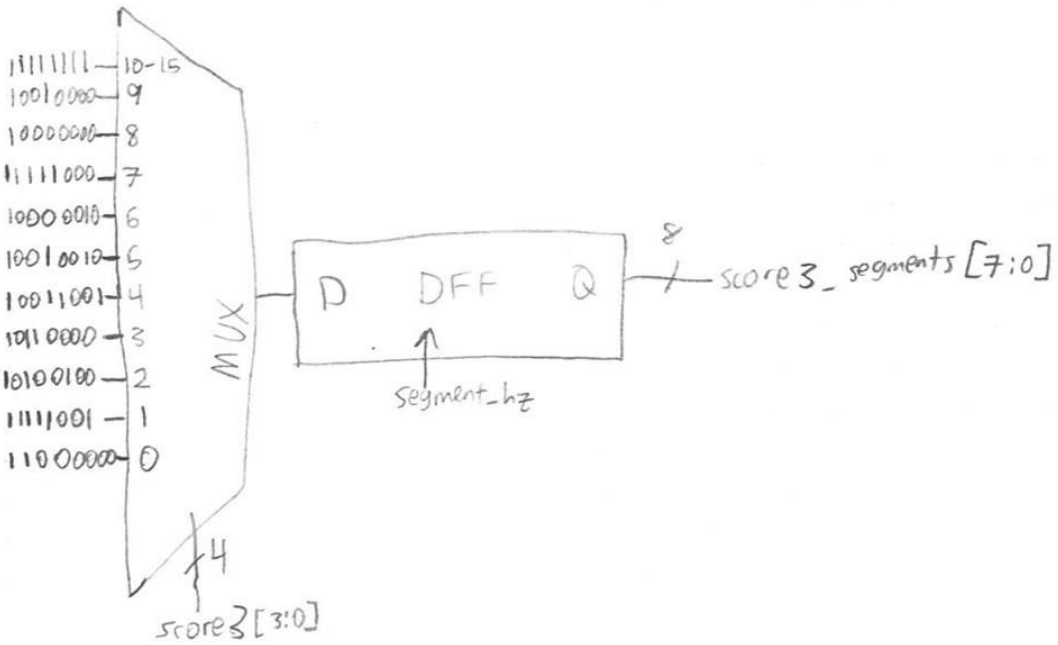

*Figure 6: Digit value to segment encoding for a single digit.* A multiplexer is used to determine the segment encoding given the digit to encode, and the encoding is stored in a register. This is repeated for the remaining score digits.

Four 8-bit registers were used to store the value of the digit encodings for each of the score's digits, with another 8-bit register to store the final segment encoding for the digit being displayed and another 8-bit register to store the blank segment encoding.

First, case statements are used to determine the digit encodings for each of the digits, which are implemented using multiplexers. For the sake of brevity, only the implementation for one digit, score3 (thousands), encoding is shown in Figure 6, but the encodings for the score2, score1, and score0 are identical. The encoding is then stored into the 8-bit register for each of the digits. These registers are updated on every positive edge of the segment display clock.

Next, a 2-bit counter incrementing on every positive edge of the segment display clock is used to determine which of the four digits to display. If the dinosaur is still alive (alive has a value of 1), then the digit index and segment cycle through the four digits sequentially using the value of the counter. The high frequency of the segment display clock causes the display to appear as if all four digits are being shown simultaneously.

If the dinosaur is no longer alive, the digit index and segment are chosen in a similar cycling fashion, but this time triggered by the positive edge of the blinking clock. The blinking clock is used to determine whether or not to display the score's digits or the blank segment to give the illusion of the score blinking upon the dinosaur dying. The implementation of the logic is shown in Figure 7.

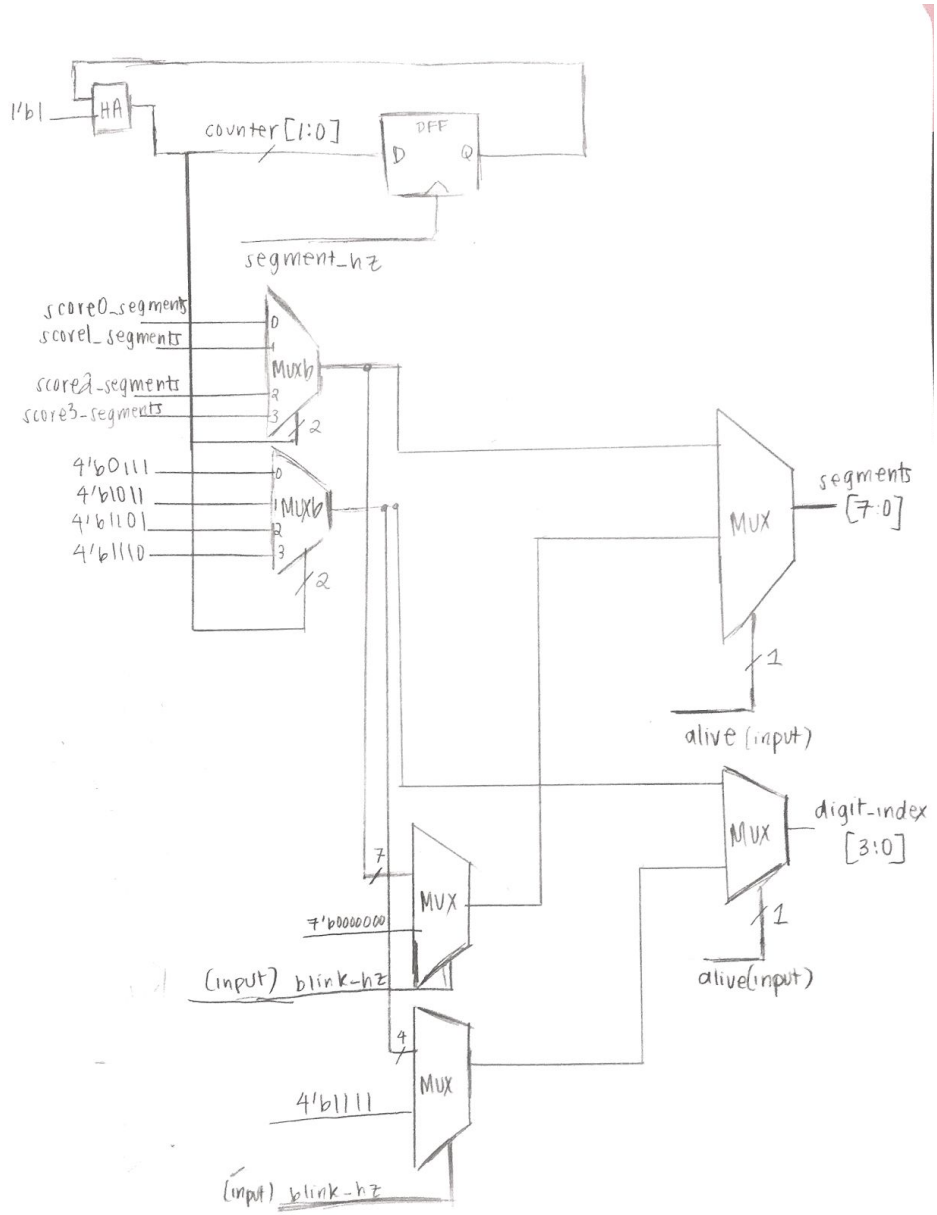


*Figure 7: Seven Segment Module.* A 2-bit counter is incremented once every clock cycle on a very fast clock, and one of the four digits is displayed each cycle if the player is alive. Since the clock is so fast, this appears as a simultaneous displaying of four digits to the human eye. If the player is not alive, on a slower blinking clock, *blink_hz*, we choose whether or not to display the score by setting the *digit_index* and *segments* to display nothing every blink. Note that the score inputs are from the counter module in Figure 5.

**Cloud Module**

The purpose of the cloud module is to keep track of the cloud icon's position on the screen. It takes as input the cactus clock from the clock divider and the reset signal after debouncing. Figure 8 illustrates the implementation of this module, and outputs the 10-bit cloud_h and cloud_v values indicating the cloud's top right corner position relative to the top right corner of the VGA monitor. Compared to a conventional

*xy* plane, the *h* value indicates the *x* position, whereas the *v* value indicates the *y* position, and this notation is used for all of our moving objects.

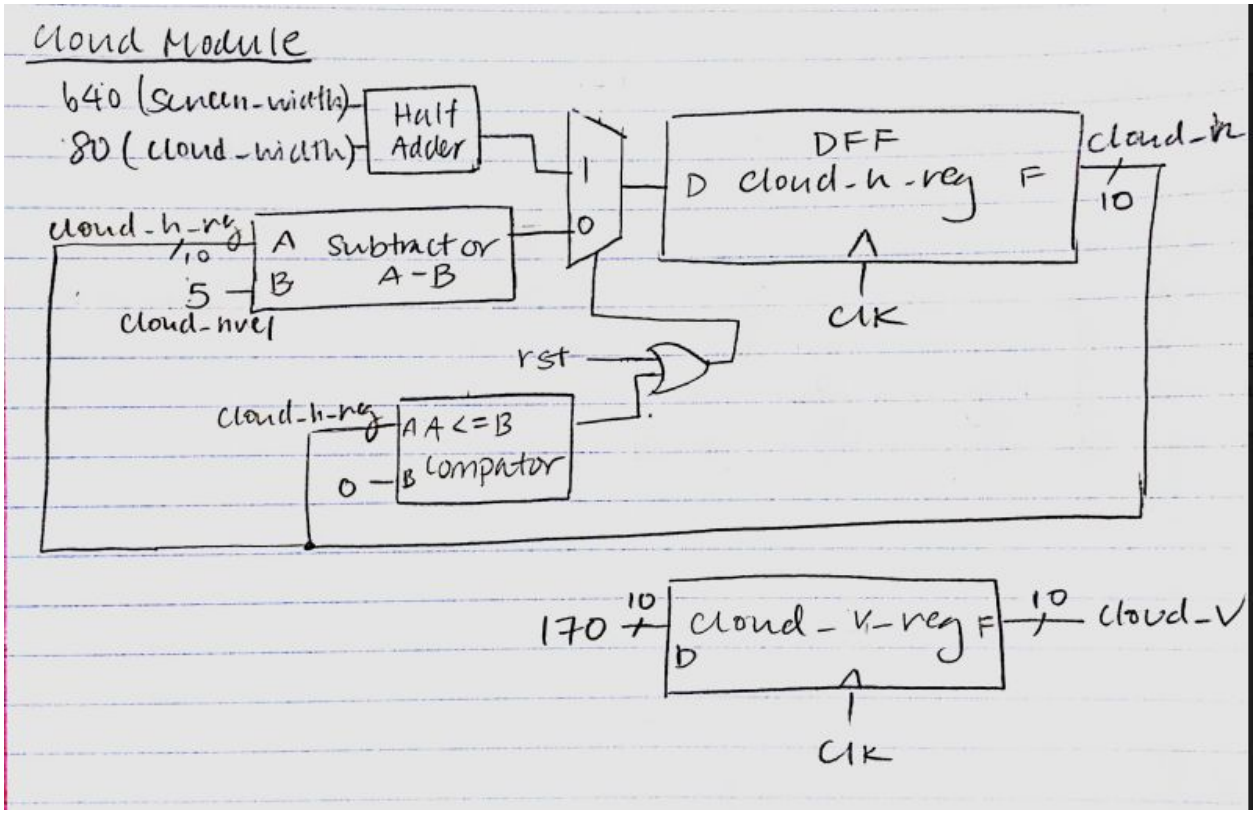


*Figure 8: Cloud module implementation*. The cloud_h and cloud_v values determine the position of the cloud on the VGA monitor.

The module uses a set of fixed constants as part of its calculations, including the screen width (640), cloud width (80), and cloud horizontal velocity (5). If the reset signal is triggered or the cloud has reached the left edge of the screen, the cloud is reset to the right edge of the screen by setting cloud_h to the sum of the screen width and cloud width. Otherwise, it moves left across the screen with a constant velocity, subtracting the horizontal velocity from the previous cycle cloud_h at every positive edge of the clock. The cloud_v is a constant value of 170 throughout because the cloud is not meant to move vertically.

**Dino Fields Module**

This module is responsible for keeping track and updating the position of the dinosaur icon on the monitor. The inputs to the module are the jump button signal after debouncing, clock from dino_clk, and reset signal after debouncing. It outputs the dino_h and dino_v positions indicating the position of the dinosaur's upper left corner relative to the upper left corner of the VGA monitor. The implementation of the diagram is shown below.

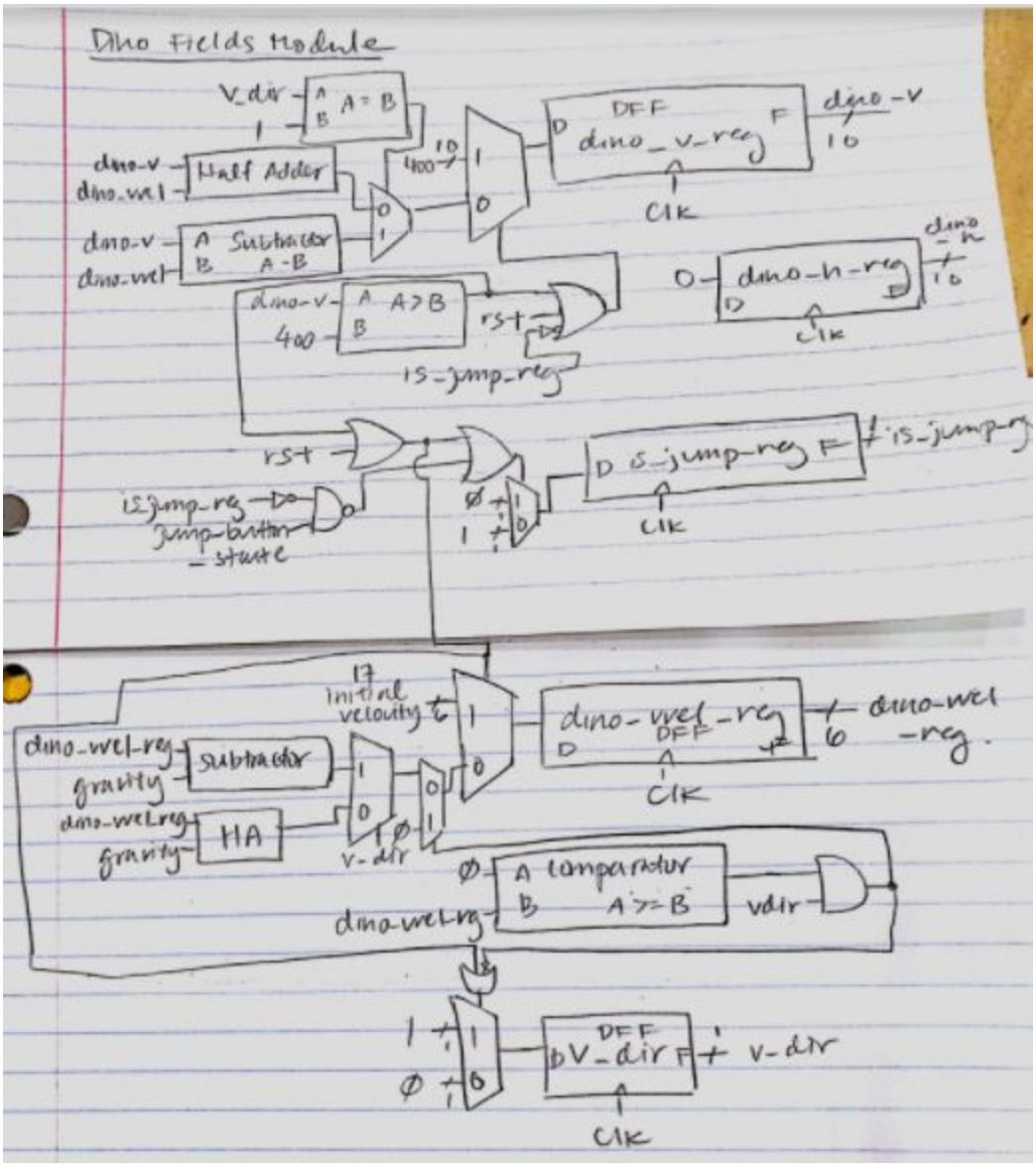


*Figure 9: Dino Fields Module*. When jumping, the dino_v position is updated based on its current velocity and direction. It remains stationary otherwise.

There are four registers that are used to keep track of the dino_v position, the jump state (is_jump_reg), vertical velocity (dino_vvel_reg), and velocity direction (v_dir). If the dinosaur's vertical position has reached below the ground or the game is being reset, the dinosaur's position is set to the ground with the vertical velocity being initialized to the initial velocity (a constant value), velocity direction set to indicate upward movement, and jump state to false. If the jump button is not pressed, the registers remain at the same values to indicate the dinosaur is stationary. The dino_h position is constantly at the left edge of the screen.

If the jump button is pressed while the dinosaur is not in the jump state, then the jump state is set to 1. Once in the jump state, the vertical velocity of the dinosaur is adjusted based on the direction of the

velocity. If the velocity is directed upward, then the magnitude of the velocity is subtracted from dino_v and the magnitude of the velocity is decreased by a fixed “gravity” amount. Otherwise, if the velocity is directed downward, then the dino_v is increased by the magnitude of the velocity and the velocity is increased by the fixed “gravity” amount. Once the magnitude of the velocity is less than or equal to 0 when the velocity is directed upward, this indicates that the dinosaur has reached the top of its jump, so the direction of the velocity is changed and the magnitude of the velocity is set to zero.

**Collisions Module**

The purpose of this module is to detect whether the dinosaur icon has intersected with either of the enemy or obstacle icons, which would end the game if true. This module takes as input the positions of the dinosaur, obstacle, and enemy (dino_v, dino_h, obstacle_v, obstacle_h, enemy_v, enemy_h) from the dinofields, obstacle, and enemy modules. It also takes the obstacle and enemy height and width, dclk from the clock divider as clock input, and the reset signal after it has been through the debouncer module. The output is a 1-bit value for the alive state of the dinosaur (1 for alive). The implementation is shown below.

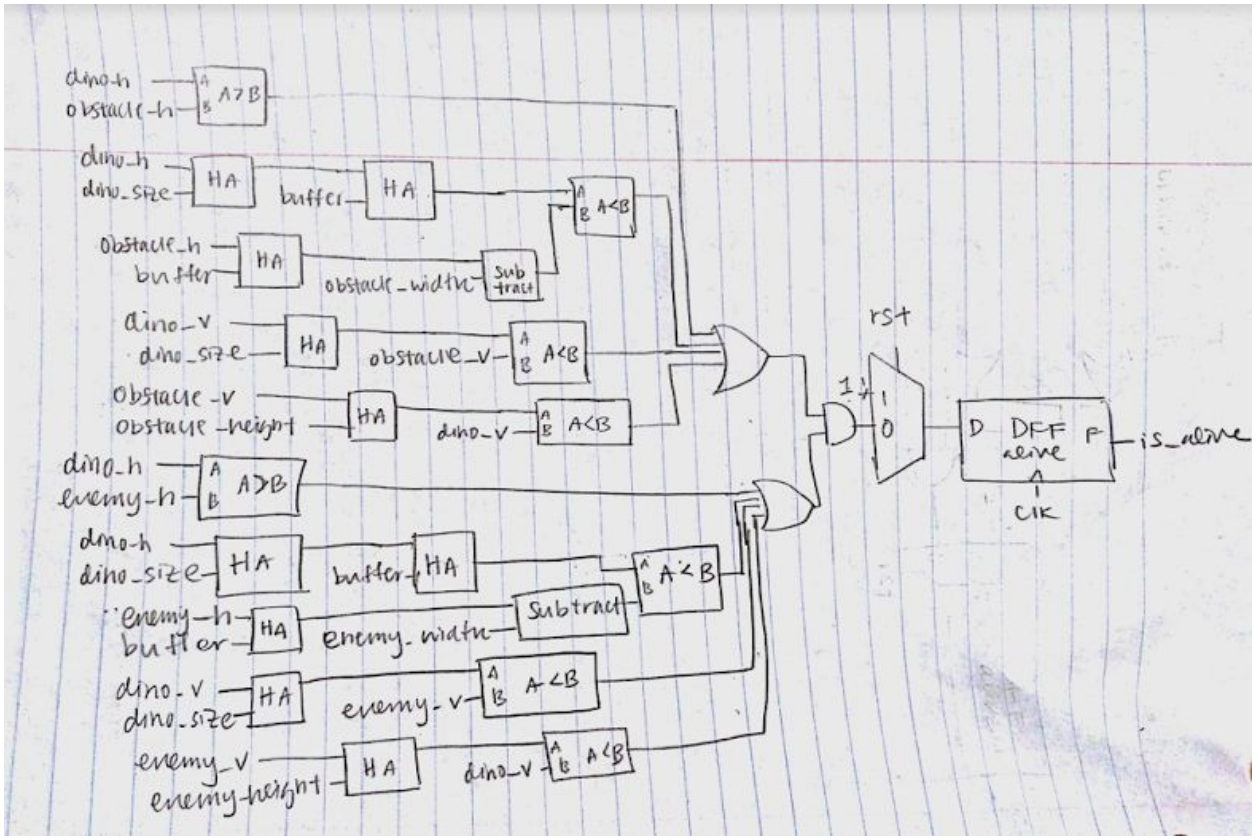


*Figure 10: Collision Module.* The enemy, obstacle, and dinosaur positions and dimensions are used to determine whether the dinosaur has collided with the enemy or obstacle.

First, if the reset signal is triggered, then the alive state is set to 1. To determine whether the dinosaur has collided with an object, there are four positions to check: above, below, left, and right of the object. If the right edge of the dinosaur is less than the left edge of the object, or if

the left edge of the dinosaur is greater than the right edge of the object, or if the top of the dinosaur is greater than the bottom of the object, or if the bottom of the dinosaur is less than the top of the object, then the dinosaur is in a position that allows it to stay alive. This logic is applied to check for both the enemy and obstacle objects, and if the dinosaur is in a valid position relative to both the enemy and obstacle, then the alive state remains 1. Otherwise, a collision has detected, at which point the alive state is set to zero.

Note that the collisions module takes object coordinates from the dino fields module, the obstacle module, and the enemy module in order to detect a collision. Furthermore, its output, the alive state, is used in the counter module to determine whether or not to increment the score, and in the seven-segment display module, in order to display the final score at a blinking frequency.

**Obstacle Module**

The purpose of the obstacle module is to keep track of the obstacle icon's position on the screen. It takes as input a reset signal after debouncing and a clock signal from the clock divider to determine the frequency at which the obstacle should move across the screen. Additionally, it takes in a random obstacle number generated by the LFSR module that determines which type of obstacle it is, and an 8-bit random obstacle time that determines the wait-time before the obstacle enters the view from the right side of the screen.

If the reset signal is triggered, then the obstacle is reverted back to the initial state, which is known as the "wait_state," meaning that the obstacle is not yet on the screen and is waiting until it reaches the "move_state." A state diagram is shown in Figure 11. If the obstacle is in the wait state, then prevent the obstacle from appearing on the screen until a counter reaches a randomly generated amount of waiting time. The implementation of the wait state is shown in Figure 12, as the wait state is negated once the counter reaches the random value. When the counter reaches the time, the obstacle is no longer in the wait state and is instead in the move state.

In the move state, move the obstacle to the left by updating its horizontal coordinate. The implementation of the move state is shown in Figure 13. If the obstacle has reached the left edge of the screen, then it is out of bounds, in which we perform one of three actions based on the ⅓ probability given by the random obstacle_num modulus three: if 0, we reformat the coordinates and dimensions to be a cactus; if 1, we reformat the obstacle to be a pterodactyl flying a bit higher in the air; otherwise, we reformat the obstacle to be a pterodactyl flying lower to the ground. We then return to the wait state, and repeat the process again by selecting a new random number and waiting to display a new obstacle on the screen upon the counter reaching the new value.

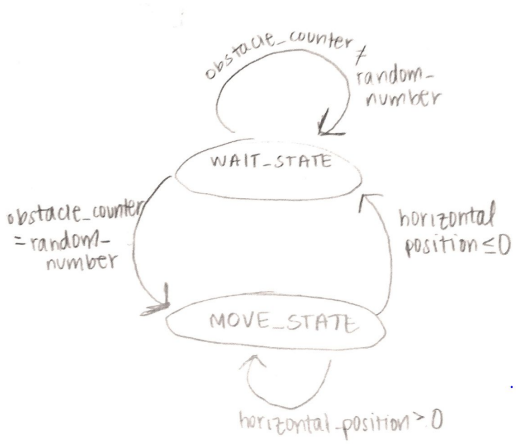


*Figure 11: Obstacle Module State Diagram.* The state diagram consists of a "wait_state," when the obstacle is waiting until it appears on the screen, and a "move_state," when the obstacle is moving from the right of the screen to the left of the screen, toward the dinosaur.

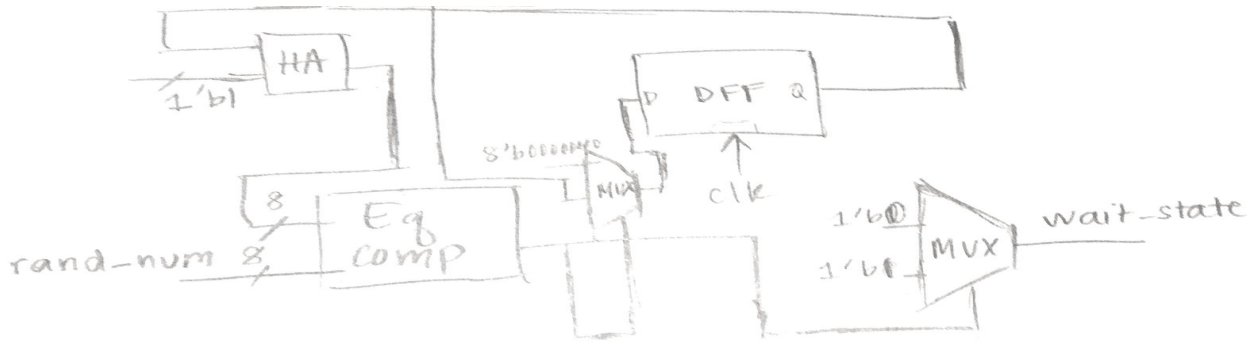


*Figure 12: Schematic of Wait State.* The wait state consists of an 8-bit counter being continuously incremented until it reaches the number that was randomly generated by the LFSR module. If the value is reached, the wait state is negated and the obstacle is now in the move state.

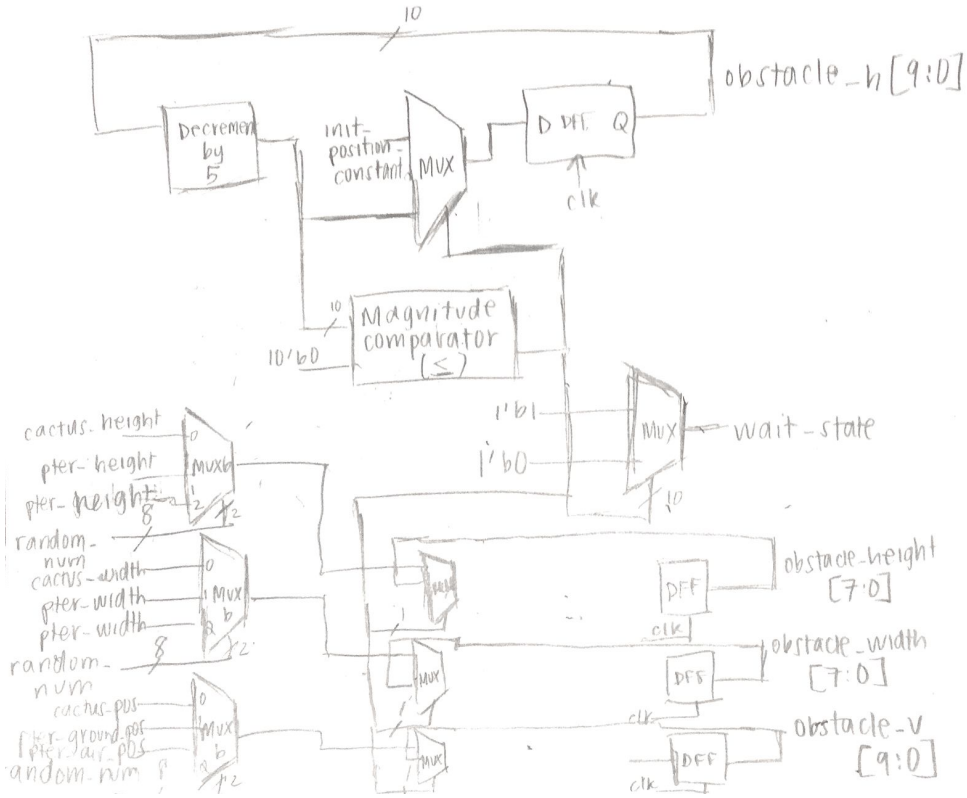


*Figure 13: Schematic of Move State.* In the move state, the obstacle is moved to the left by 5 pixels every clock cycle. If the left edge of the screen is reached, then a new random number from the LFSR module is used to select a new random wait time until the obstacle appears; the obstacle is now set to be in the wait state. A new random number is also used to decide whether the obstacle height, width, and vertical position reflect those of a cactus, a pterodactyl in the air, or a pterodactyl on the ground.

The obstacle module outputs the obstacle's horizontal and vertical coordinates denoted by the 10-bit obstacle_h and obstacle_v values indicating the obstacle's top right corner position relative to the top right corner of the VGA monitor, as well as its horizontal velocity (obstacle_hvel), the height (obstacle_height), the width (obstacle_width), and whether or not the obstacle is currently in a wait state (wait_state). As described in the cloud module, the *h* value indicates the *x* position, whereas the *v* value indicates the *y* position in a conventional *xy* plane.

There are 6 registers that are used to keep track of the obstacle_height, obstacle_width, obstacle_h, obstacle_v, obstacle_hvel, and wait_state. Initially, these registers are formatted so that the obstacle is at first a long, upright cactus in the wait state that is placed just to the right of the display view (around 640 pixels across) so that it is not visible on-screen.

In summary, the obstacle is initialized as a cactus that moves to the left towards the dinosaur and then randomly becomes one of three obstacles to move towards the dinosaur, each iteration of the obstacle waiting a specific, randomly-selected wait time before appearing on the screen to give the illusion of randomly-timed obstacle spawning.

**Enemy Module**

The enemy module is used to generate a second obstacle (also referred to as an "enemy") for the dinosaur. The enemy module performs a nearly identical function as the obstacle module, as it takes the same inputs, outputs an obstacle's coordinates, and also waits for a random amount of time before moving the obstacle toward the dinosaur. The key difference is that the enemy module also takes in the obstacle position (from the obstacle module) so that the enemy does not appear too close to the other obstacle. A state diagram is shown in Figure 14.

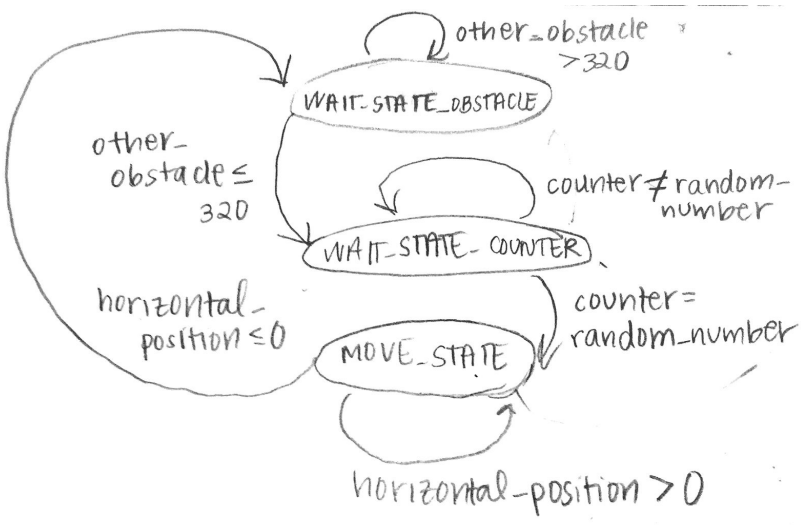


*Figure 14: State Machine for Enemy Module.* The enemy module has one extra state than the obstacle module. The initial state of the enemy module is "wait_state_obstacle," which means that it is waiting for the other obstacle to go past a certain checkpoint of the screen. After the checkpoint is reached, the state shifts to the "wait_state_counter" state which is analogous to the wait state in Figure 11; the enemy waits for a randomly generated amount of time and increments a counter until that random value is reached. When the counter reaches the value, the "move state" is reached, and the enemy moves from the right until the left end of the screen. At the end of the screen, the obstacle goes back to "wait_state_obstacle."

The implementation of the "wait_state_counter" state in enemy is the same as the "wait_state" in obstacle, shown in Figure 12. Moreover, the implementation of "move_state" in enemy is also

identical to the “move_state” in obstacle, shown in Figure 13. The difference is the initial state, “wait_state_obstacle.” This initial state is illustrated in Figure 15.

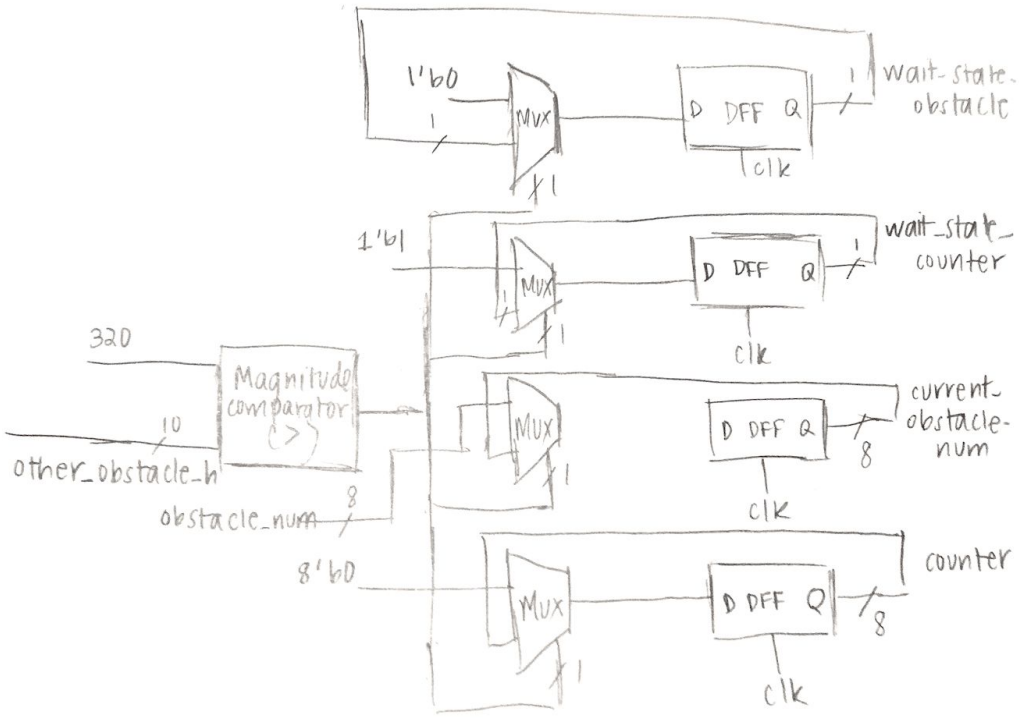


*Figure 15: Wait State Obstacle for Enemy.* The initial state of the enemy module is “wait_state_obstacle,” which means that the enemy is waiting for the other obstacle to go past a certain checkpoint of the screen. When this checkpoint is reached, the state variables are updated to reflect the new state. The counter is set to 0 and the current_obstacle_num is set to the random number from the LFSR module, in preparation for wait_state_counter state, which is analogous to the wait state in Figure 12.

Another difference between enemy and obstacle is the initial parameters; instead of starting off as a cactus, the enemy starts off as a pterodactyl on the ground, which a triggered reset signal will cause the enemy to revert to. The reason for adding the enemy module module was to allow more variety in the obstacles presented and to also not have the player wait too long before another obstacle appears. Simultaneously, this enemy module takes in the position from the obstacle module so that the random position of the enemy is not impossible for the user to clear given the position of the obstacle.

**LFSR Module**

This module is used as a pseudo-random number generator, and the code was taken directly from the resource mentioned in the references section (Reference #1). It takes as input cactus_clk signal from the clock divider, an enable signal always set to 1, and reset signal from the

debouncer. The output is an 8-bit pseudo-random number. Since we did not write the implementation of the module, we will treat it as a black blox.

**VGA Display Module**

This module is responsible for updating the 640x480 size VGA monitor display. It takes as input the dclk from the clock divider, reset signal after debouncing, obstacle, dinosaur, enemy, cloud positions, obstacle and enemy dimensions, and alive state of the game. It returns as output the horizontal sync output (hsync) and vertical sync output (vsync) along with the 3-bit red, 3-bit green, and 2-bit blue VGA outputs. The first part of the module deals with updating hsync and vsync, shown below.

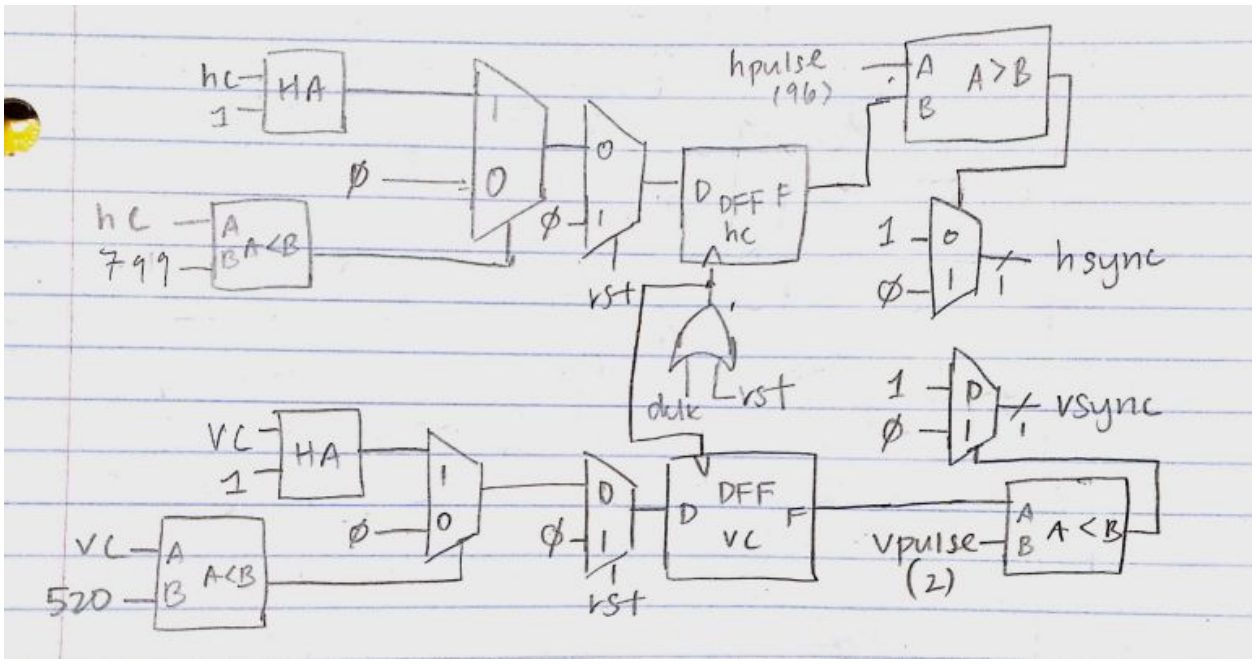


*Figure 16: Section of VGA module that updates hsync and vsync.* The updates are based on the hc and vc registers, which keep track of the pixel position on the display grid.

The program moves across the frame of pixels from left to right, top to bottom, starting from the upper left corner (0, 0) by using two registers, hc and vc, to keep track of the horizontal and vertical position on the grid. As shown in the figure above, each clock cycle updates hc by increasing it by one until it reaches the right edge of the display, at which point it resets to zero. At this point, vc is incremented by one. Once vc reaches the end of the frame, it is also reset to zero. Hsync and vsync outputs are assigned based on the values of hc and vc compared to fixed parameters hpulse and vpulse.

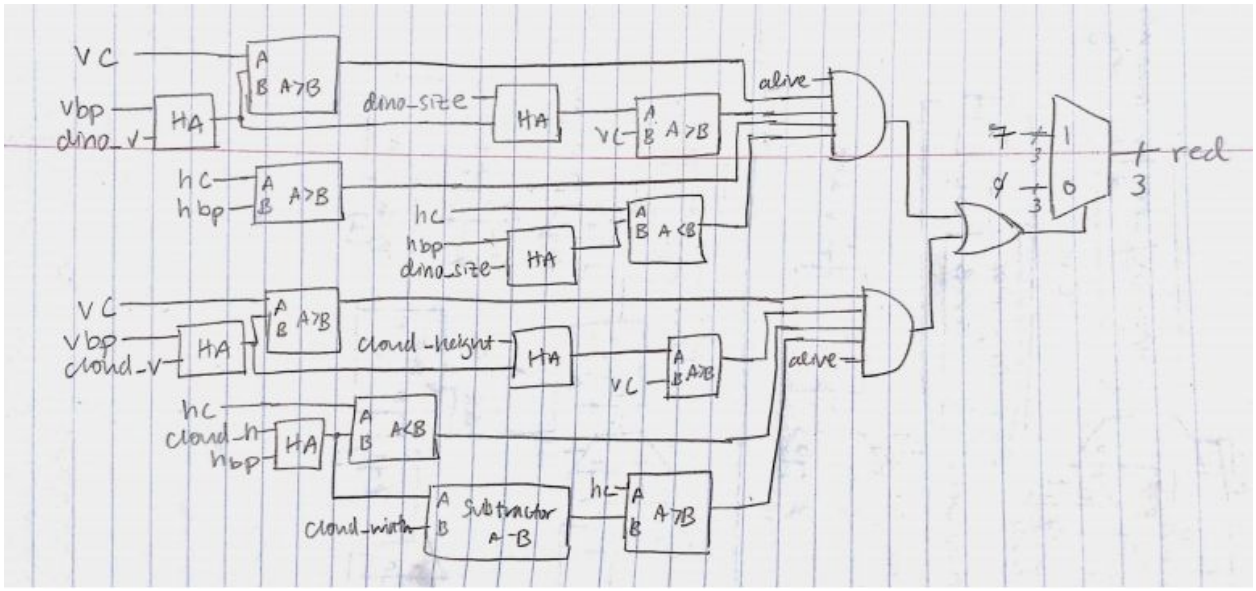


*Figure 17: Logic for determining the VGA red output*. The (hc, vc) positions determine which pixels on the screen should be have red color.

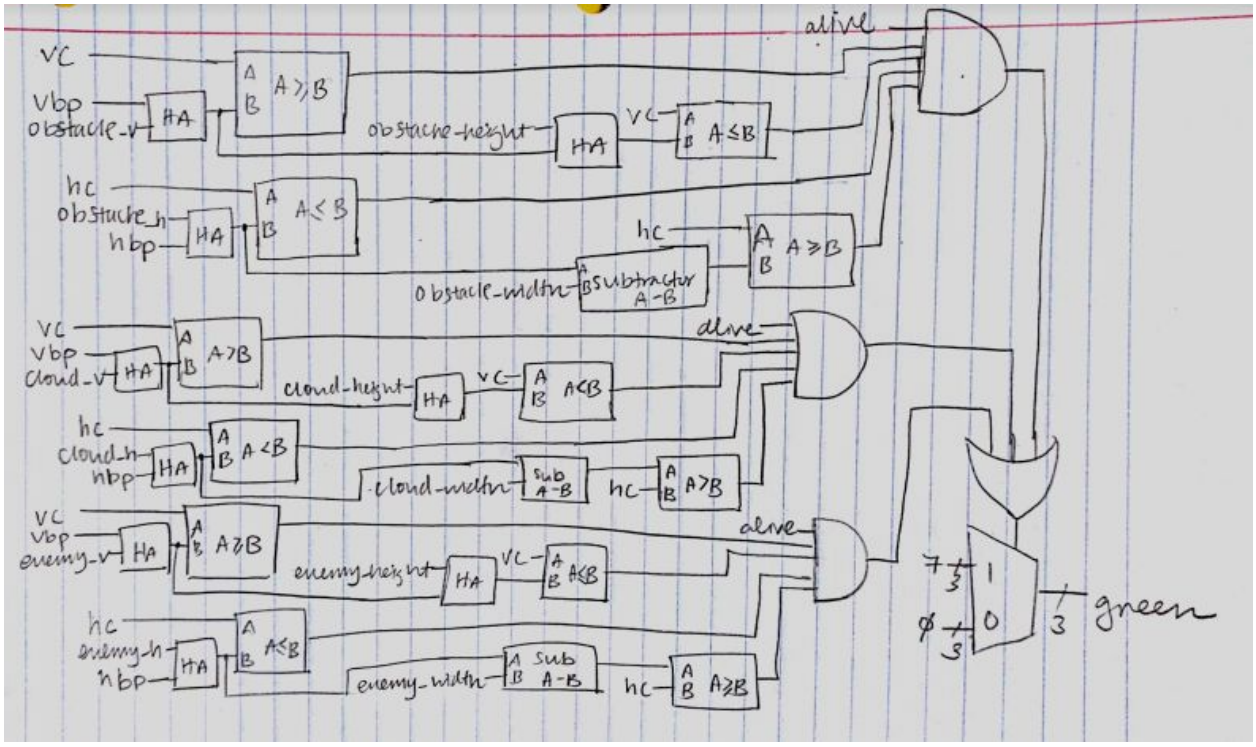


*Figure 18: Logic for determining the VGA green output*. The (hc, vc) positions determine which pixels on the screen should be have green color.

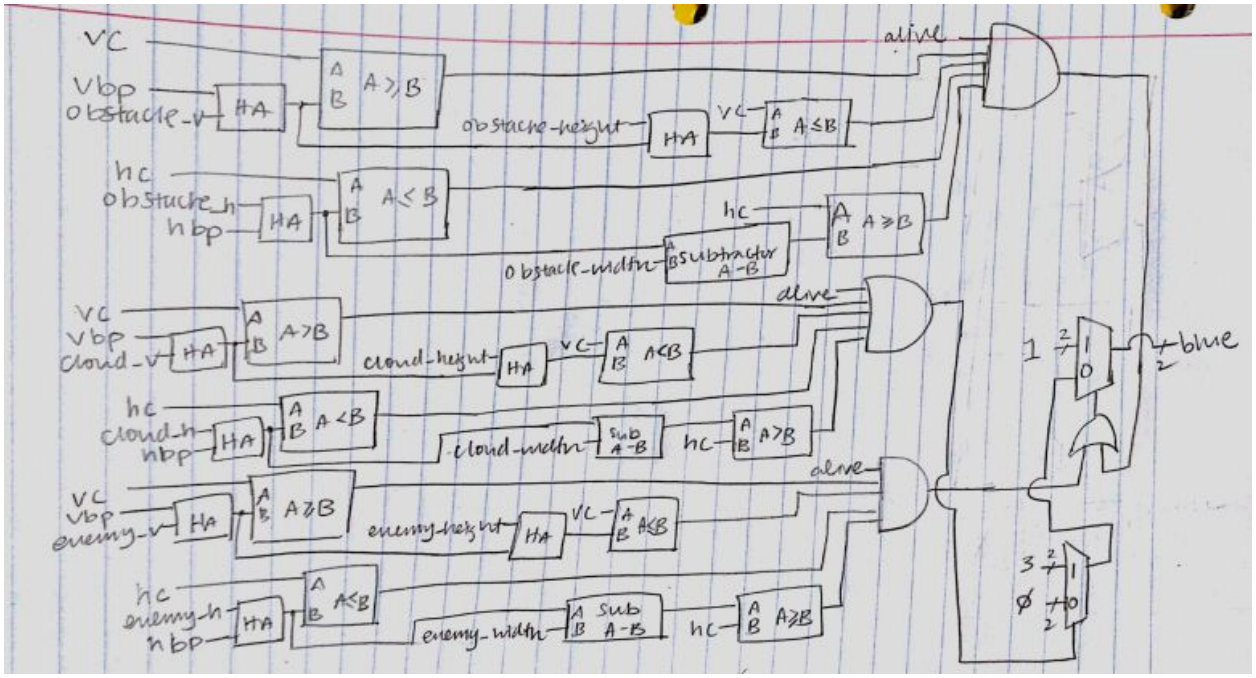


*Figure 19: Logic for determining the VGA blue output*. The (hc, vc) positions determine which pixels on the screen should be have blue color.

The red, green, and blue colors are determined based on the current (hc, vc) positions on the pixel grid and if the game is in the alive state. If the (hc, vc) is within the dinosaur's position, then the coloring is purely red. If the (hc, vc) is within the cloud's position, then all the colors mixed to produce white. If the (hc, vc) is within the obstacle or enemy's position, then the coloring is part blue, part green. If the (hc, vc) is within the ground area, then the coloring is a combination of red and green. Otherwise, if the (hc, vc) is outside these ranges or the alive state is zero, then the pixel coloring is zero for a black color.

**Testing Documentation**

*Debouncer Module*

This module could not be tested with simulation. After implementing the design on the FPGA board, we tested that the button pushing registered properly by verifying the output on the seven-segment display and the output on the screen.

*Counter Module*

We test the counter module for proper incrementation of each of the score's digits under various states. The inputs to the counter module were adjusted to change the counter state, and the score's digits (denoted from left to right as $S_3$, $S_2$, $S_1$ $S_0$) were observed on the seven-segment display to verify correctness.

*Table 1: Test Cases for Counter Module*

| State | Expected Output | Passed? |
|---|---|---|
| Increments values correctly | $S_0$ increases thrice every second (3 Hz), and wraps around to 0 once it reaches 9. $S_1$ increases when $S_0$ reaches 9, and wraps around once it reaches 9. Then, $S_2$ increases until it reaches 9, at which point $S_3$ increases. In other words, "carry over" should work when incrementing the score. | Yes |
| Score remains fixed when player dies | When the player dies (indicated by the alive signal being 0), the value the four-digit score contains should remain fixed and no longer increment. | Yes |
| Reset | When the reset signal is high, the score should reset back to 0000. | Yes |

*Seven Segment Display Module*

This module was tested for proper segment encoding given a digit, along with testing for displaying the correct digit at the appropriate time.

*Table 2: Test cases for Seven Segment Display module*

| State | Expected Output | Passed? |
|---|---|---|
| Normal counting state only when player alive, and reset state | Cycle through displaying $S_3$, $S_2$ $S_1$, and $S_0$ rapidly. Score digits appear to display simultaneously. | Yes |
| Blinking game over score | When the player dies (indicated by the alive signal being 0), every other cycle, instead of displaying $S_3$ through $S_0$, do not display anything. This causes the digits to appear to blink. The score should be fixed. | Yes |
| Reset | Display resets back to 0000. | Yes |

*Clock Divider Module*

This module was tested for proper division of the master clock frequency to attain the clock frequencies desired. The output frequencies were mainly experimented with through trial-and-error so that the display would be viewable and the game would be playable. We first verified that the display frequency was correct by ensuring that the NERP_demo from the Lab manual would display on the monitor. The other frequencies for moving obstacles were tested

after implementing those modules, by verifying that the frequencies were suitable for gameplay by looking at the VGA display.

*Table 3: Test cases for Clock Divider module*

| State | Expected Output | Passed? |
|---|---|---|
| VGA Display Frequency Output | When pressing the button on the monitor to view VGA output, the screen should display the colored bars from the NERP demo (at this stage, the NERP Demo from the lab manual was used for testing the display, prior to adding our own graphics).[3] | Yes |
| Change obstacle and enemy speeds | Enemy and obstacle clocks should be faster if the level is 1. This was tested after implementing the scrolling screen. | Yes |

*Cloud Module*

The cloud module was tested by observing its displayed behavior when moving across the VGA display screen.

*Table 4: Test Cases for Cloud Module*

| State | Expected Output | Passed? |
|---|---|---|
| Moves left at constant speed | The cloud should move at a constant horizontal speed to the left. | Yes |
| Cloud wraps around | When the cloud moves off the left edge of the screen, the cloud's position should wrap back around to the right edge as it was initially placed. It should then resume moving at constant speed to the left. | Yes |
| Reset | When the reset button is triggered, the cloud should be placed back at the right edge of the screen. | Yes |

Likewise, the subsequent dino fields, obstacle, enemy, collisions, and VGA modules were tested similarly by observing their displayed behaviors on screen in response to the player's actions, the environment, etc.

*Dino Fields Module*

*Table 5: Test Cases for Dino Fields Module*

| State | Expected Output | Passed? |
|---|---|---|
| Stationary | The dinosaur should be stationary at the left edge of the screen on the ground when the player doesn't do anything. | Yes |
| Jump | When the player chooses to jump, the dinosaur should jump vertically with effects to simulate gravity affecting its jump. It should slow as it reaches apex of jump and then speed up as it descends. It should then land back on the ground and remain stationary. The dinosaur cannot jump if already in the air. | Yes |
| Reset | When the reset button is triggered, the dino should be placed back at the left edge of the screen on the ground. | Yes |

*Obstacle Module*

*Table 6: Test Cases for the Obstacle Module*

| State | Expected Output | Passed? |
|---|---|---|
| Initial form | Generate a cactus at right edge of screen as first obstacle. | Yes |
| Horizontal movement | The obstacle should crawl to the left at the given clock speed towards the dinosaur. | Yes |
| Obstacle wraps around | When the obstacle reaches left edge of screen, it should wrap around and be placed past the right edge to get ready to crawl again. | Yes |
| Obstacle random form | After the obstacle goes off screen, it should after wraparound choose a random form: cactus, low pterodactyl, or high pterodactyl. | Yes |
| Obstacle random wait time | After choosing the new form and wrapping around, the obstacle must wait a random amount of time before appearing on the screen. | Yes |
| Reset | After reset is pressed, the obstacle should be placed back at the starting point as a cactus. | Yes |

*Enemy Module*

The test cases for enemy module are exactly the same as the obstacle module, except for some slight differences with the random wait time, initial form, and reset. Only these will be included in the table.

*Table 7: Test Cases for Enemy Module*

| State | Expected Output | Passed? |
|---|---|---|
| Initial form | The enemy initial form should be a low pterodactyl placed to the right of screen. | Yes |
| Enemy random wait time | The enemy needs to wait before entering the screen a random amount of wait time, plus the obstacle's wait time to avoid them both entering at too close of a time. In other words, the enemy should not appear too close to the obstacle. | Yes |
| Reset | After reset is pressed, the enemy should be placed back at the starting point as a low pterodactyl. | Yes |

*Collisions Module*

*Table 8: Test Cases for Collisions Module*

| State | Expected Output | Passed? |
|---|---|---|
| No collision | If the dinosaur is not touching the obstacle or enemy, the dinosaur should still be alive, the score increasing, and game still displaying and moving. | Yes |
| Collision | If the dinosaur's square overlaps the rectangle of either the obstacle or enemy, the dinosaur should be declared no longer alive. This should work for all three obstacle types (high pterodactyl, low pterodactyl, and cactus) and should work regardless of whether the collision point is at the left or right side of the obstacle and the dinosaur. | Yes |
| Reset | When the reset button is triggered, the alive state should be reset back to alive and restart a fresh game with assumed no collision. | Yes |

*LFSR Module*

Because we used LFSR module as a black box, we did not test it other than that it generated random numbers to affect the enemy and obstacle entities as described above.

*VGA Display Module*

*Table 9: Test cases for VGA Display Module*

| State | Expected Output | Passed? |
|---|---|---|
| Display Ground | Brown strip of ground about 20 pixels tall off the bottom of the screen should be visible. | Yes |
| Display Dino | Red square of size 40x40 should be displayed on the left edge of the screen on top of the ground. Dino should also move vertically based on its position when jumping. | Yes |
| Display Cloud | White rectangle of size 80x30 (format is horizontal width x vertical height) should be displayed based on the cloud's position. | Yes |
| Display Obstacle/Enemy | Green rectangle should be displayed according to its form and at the obstacle's/enemy's position:<br>● If cactus: 30x80 green rectangle should be displayed with base on the ground.<br>● If high pterodactyl: 60x30 green rectangle should be displayed with its base 35 pixels off the ground.<br>● If low pterodactyl: 60x30 green rectangle should be displayed with its base 5 pixels off the ground. | Yes |
| Dino Dead screen | If the dinosaur is dead, none of the above shapes should be displayed and a completely black screen should fill the display. | Yes |
| Reset | Reset the VGA display to the initial display: black background with brown ground, dinosaur to the left, and moving entities (cloud, obstacle, enemy) placed off the right-edge of the screen out of view. | Yes |

*Overall System ~ Main Module*

Finally, the overall system was tested by playing the game and ensuring that the different states were as desired on the seven-segment display and on the monitor. Since the dino fields, obstacle, enemy, and collisions module all work closely together, the tests for those modules were also part of the overall system test. Table 10 summarizes the overall system tests, many of which are described in more detail in the individual corresponding modules.

*Table 10: Test Cases for Overall System.*

| State | Expected Output | Passed? |
|---|---|---|

| | | |
|---|---|---|
| Reset | The game resets to the initial state (see the criteria for the other modules to see what should happen when reset is pressed: the score should start again at zero, and the obstacles should be positioned in the starting game state). | Yes |
| Jump | The dinosaur jumps off the ground only upon the initial press and nothing else (e.g. noise). Pressing the button for longer should not affect jump height. | Yes |
| Dinosaur Alive | The dinosaur remains alive until it collides with an obstacle on the monitor. Simultaneously, the score should increment at 3 Hz on the seven-segment display. | Yes |
| Dinosaur Dead | Upon collision, the score should no longer increment. A black screen should be displayed. | Yes |
| Speedup | Once the game reaches a score of 40, the frequency at which the obstacles move increases. | Yes |
| Move Cloud and Hazards | Cloud and hazards horizontally scroll across the screen at their clock rates, wrapping back around when it goes off the left edge of screen. | Yes |
| Randomly choose Obstacle/Enemy form | After the obstacle or enemy wraps back around after going off the screen, choose a new form for it (cactus, low pterodactyl, and high pterodactyl) to present a new hazard to the dinosaur. | Yes |
| Randomly Generate Obstacles | Obstacles should occur at random times and spacings, with a minimum spacing between obstacles such that it is indeed possible for the user to clear both. | Yes |

**Conclusion**

We achieved our goal of creating a simplified version of the Google Chrome *T-Rex Offline Game* on the FPGA board, fulfilling the all of the specifications outlined in the introduction and grading rubric. At a high level, there are 13 key modules contained within a main module, working together to organize the gameplay. For controls and gameplay mechanics, we used the debouncer to debounce the jump and reset raw button inputs, the clock divider to generate different clock frequencies, a counter to count up the player's score, a collisions module to determine if the dinosaur collided with the enemy or obstacle, and a seven segment display module to convert the score's digit values into segments for display, as well as a VGA display module to display on the VGA screen the game board. For the game's entities, we implemented the dino fields module to update the states and variables of the dinosaur, the cloud module to update the position of the background cloud moving across the screen, and the obstacle and

enemy modules to update the states and variables of the moving obstacle and enemy to act as hazards to the dinosaur. Lastly, the LFSR module was used to generate random numbers to determine the hazards' forms and wait times.

While it was an enjoyable project, one of the most frustrating difficulties we repeatedly encountered was the way VGA display worked with its horizontal and vertical coordinates. This usually had to do with negative coordinate positions that caused objects to suddenly disappear or not appear in their right positions. For example, because hazard entities would frequently move across and then off the screen without any clean way of tracking their VGA coordinates, the enemy or obstacle modules would have to subtract from the current horizontal coordinate to move them to the left. But if this subtraction ever caused the value to drop below zero, the integer bit-wise wraparound would cause the object to jump off the screen or other non-desired behavior. To fix this, we had to use the upper right corner of the enemy and obstacle, as well as add various buffer constants to our inequalities to ensure our calculations never dealt with negative numbers.

One improvement we could make to our game is to implement the VGA to display the actual sprites and shapes of all entities on the screen; instead of squares and rectangles, the display shows an actual T-Rex, Pterodactyl, cactus, etc. from the original game. Also, we could perhaps implement a scrolling ground as well to give a more realistic feel of the dinosaur running. But given the brevity of time to work on the project, we instead focused on the game's functionality and mechanics, for which solid-colored rectangles proved sufficient for this simplified version of the Dino Run game.

**References**


1. Tala, Deepak Kumar. Random Counter (LFSR) (ver. February 2014). Retrieved from http://www.asic-world.com/examples/verilog/lfsr.html on December 6, 2017.
2. UCLA Computer Science Department. CS M152A Lab 3 (ver. December 2017). (Univ. California Los Angeles, Los Angeles, California).
3. NERP demo used as VGA Display Module framework. Retrieved from https://www.element14.com/community/thread/23394/l/draw-vga-color-bars-with-fpga-in-verilog in November 2017.