Lab 7

Jose Santiago

3/10/20

Section 1D

Description: The purpose of this lab is to design a sequential circuit to program our Basys 3 board to play the game “Rainbow Road.” The Basys board has a VGA output to enable a The objective of the game is to keep a car on the Rainbow Road. The road randomly generates at the top of the screen as the car drives up the road. The buttons btnL and btnR on the Basys board control the left and right steering of the car. As long as any portion of the car remains on the road, the game continues. The score increments as long as the car remains on the road as well. The game ends once the car fails to make contact with any of the road. The difficulty of the game can be increased by using switches 4-6 to dictate the width of our road segments.

Results & Methods: This lab required a hefty amount of planning. A lot of the design was left for us. Although this made this lab especially daunting, the first hurdle to overcome was the VGA control module.

<u>VGAcontroller</u>: The purpose of the VGAcontroller module is to allow us to populate our monitor with the appropriate pixels for the Rainbow Road game. To do this we need to output an hsync and vsync signal, as well as an h_pixel_address and v_pixel_address. The manual includes the VGA color outputs in the VGAcontroller along with the previously mentioned outputs. Within this design I opted to create a color module to output the desired color outputs for our current pixel address. In order to keep track of the horizontal and vertical pixel addresses for our screen, we can use two 16-bit counters from our previous lab. We can create new modules for both the horizontal and vertical counter as the modularity in our design can help make the design more straightforward.

<u>H-counter</u>: The horizontal address counter was the first module to design. This module has two inputs, a clock(clk) and an enable(ce). The module also has three outputs consisting of the hsync signal, h-address, and v-count signal. The enable tells our horizontal counter when to count-up. To begin this design, call an instance of our 16-bit counter. The horizontal counter must count a pixel width of 800 before rolling over. To implement this functionality input a 16-bit zero value to the 16-bit D port of our counter instance to be loaded every time we want to roll over. The load signal for our counter instance should be enabled when we receive the ce signal and we have reached the max value of 799(indexing from 0). This rollover signal sent to the ld input port is also the v-count output. Every time we are rolling over it is necessary to tell the v-counter to count up. The up port of the 16-bit counter instance is connected to the ce input port of our H-counter module. The clock port of our 16-bit counter instance is also connected to the clock input port of the module. The down port of the counter is grounded because counting down isn’t necessary in this module. The UTC and DTC outputs of our counter instance are sent to wires that are local to the module. These outputs aren’t necessary either. The 16-bit Q output of the counter instance is sent directly to the 16-bit output port of our module, h-address. To assign the final output, hsync, we use the comparison operators to set it low from pixels 655 to 750(655<= h-counter <=750).

After completing the horizontal address counter, the next step is to create the vertical address counter.

<u>V-counter</u>: The vertical address counter has two inputs, a clock(clk), and an enable(ce). The module also has two outputs, the v-address and vsync outputs. Like with the H-counter we call an instance of our previously defined 16-bit counter within this module. The up signal is connected directly to the ce input port. The down signal is grounded since we don't need to enable counting down at any time. The ld port is connected to a local wire named rollover. This wire is high when the maximum value of our V-counter, 524(indexing from zero), is reached and the ce signal goes high. This port tells our module when we want to reset. The D input port of the 16-bit counter module has a 16-bit bus of zeroes for the reset function. The clock input to the counter instance is connected to the clock input for our v-address counter module. The UTC and DTC outputs are sent to local wires and are left untouched after as they aren't needed outside of it. The 16-bit output of the counter is sent to the 16-bit output v-address. The finaly output, the vsync signal, is low for two pixels of the 525 total v-counter address range(489-490). We use comparison logic to assign this output signal(489<= v-address <=490).

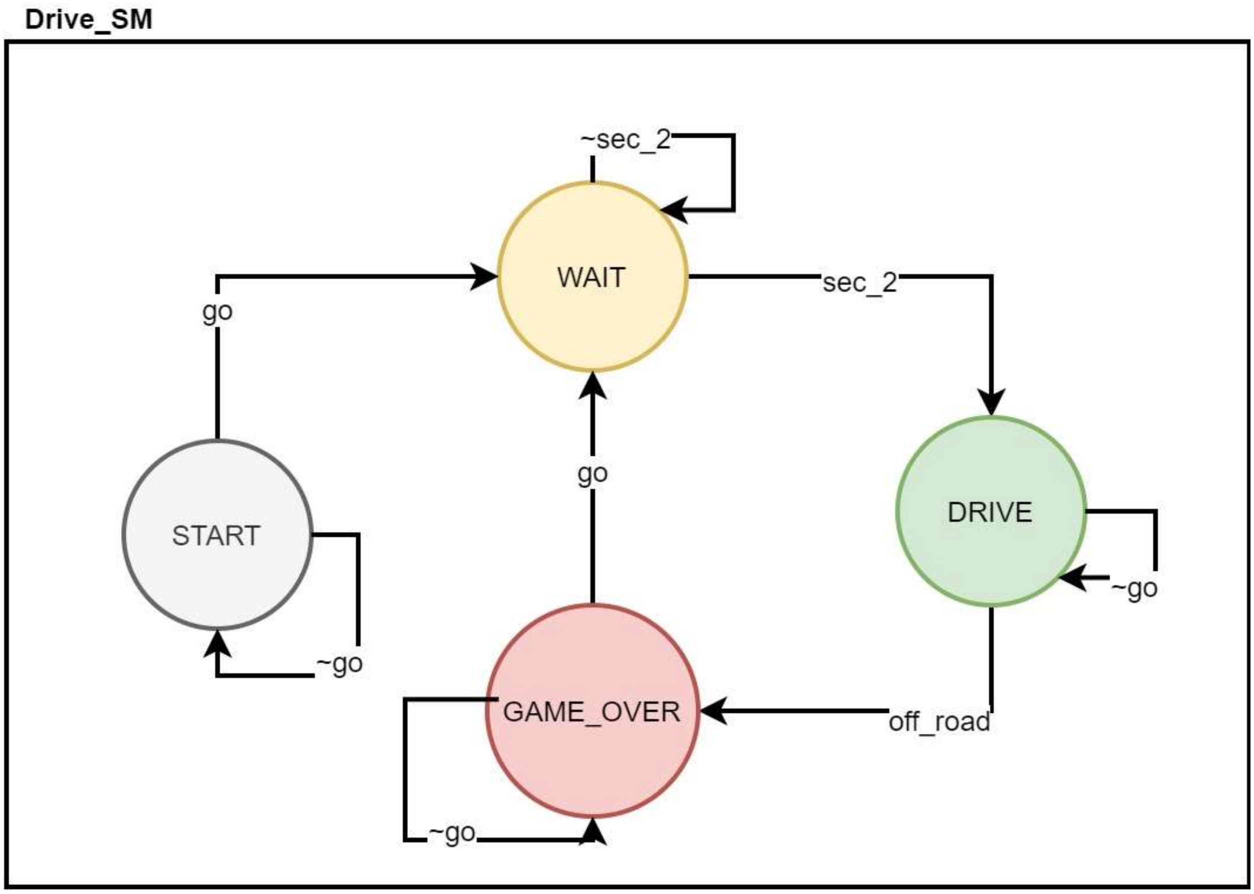
After completing the V-counter the VGAcontroller is almost complete. Call the H-counter and V-counter instances and connect the appropriate outputs to the VGAcontroller. The h-address, v-address, hsync, and vsync outputs are all taken care of by the address counter modules we just designed. All we have to do is connect these outputs to the outputs of the VGAcontroller. After this there is still one more output to define. The final output to define for our VGAcontroller is the ActiveRegion. The ActiveRegion is a signal that indicates whether our current address is within the addresses range visible on the VGA display. This signal is important to ensure we don't try to populate the nonexistent pixels with color. To do this we can use the v-address and h-address outputs to enure the ActiveRegion signal is only high while the pixel address remains on screen. The valid horizontal region is any h-address less than the outside boundary address 640. Since we index from zero, we can define the first boundary of our active reason using the comparison operator greater than or equal to to get the desired effect(h-address<=639). The vertical boundary can also be defined this way but this time with the vertical boundary of 480(v-address<=489). The ActiveRegion can de defined by AND'ing these two logical operations. This concludes the VGAcontroller module.

After getting the VGAcontroller to connect to the VGA display we can start populating the screen with the objects necessary for the game. The next module to design is the CreateObj module which outputs the ranges for the different objects we want to populate our screen.

Before creating objects we should define our control system, drive_SM.

<u>Drive SM</u>: Drive_SM is the state machine that controls our game. It has four states. The inputs to the module are: clk, go, sec_2, and off_road. The outputs of the module are drive, setup, flash_car, ResetTimer, and countdown. These output signals control our modules called in the top. The Drive_SM starts at the initial state START. Within this state we wait until the go signal is high. The go signal is defined in the top module as the edge-detected btnC input. During this transition out of the START state, the ResetTimer is set high, because time is necessary for the next state. The machine transitions into the WAIT state, where the state outputs the setup signal again, as well as the flash_car signal. The machine waits in this state untis the sec_2 signal goes high. The machine then transitions to the DRIVE state. Within this state we output the drive, score, and animate signals. We stay in this state until the off_road signal goes high. When the off_road signal goes high we transition out to the fourth and final state, the GAME_OVER state. In this state we output the flash_car signal again. The machine stays in this

state until we receive the go signal once more to transition back to the WAIT state to commence a new game. The transition into the WAIT state sets the ResetTimer signal high since we are checking for time within the WAIT state. This is the conclusion of the Drive_SM state.



CreateObj: The inputs to this module are: h-address, v-address, clk, frame, L, R, drive, setup, flash, flash_car, and size. These inputs dictate how our objects populate our VGA display. The outputs of this module are: car, off_road, and road.

The first object to create is the car since it's the simplest to define. The car is a static 16x16 pixel range located at the bottom middle of the screen. We can define the object range with two different comparison operations, one for the vertical bounds and one for the horizontal bounds((312<=h-address<=327)&&(400<=v-address<=415)). This range is defined as the car and is output from the CreateObj module to be used in the ColorModule to define the appropriate color to output to the display. After this object we move onto the more complicated objects, the road segments. The road segment objects represent the road that scrolls from the top of the screen down to the bottom. These objects have several functions to fulfill so we design a module for them.

genRoad: The genRoad module generates the seven road segments of our Rainbow Road. The inputs to this module are: clk, drive, v-address, h-address, frame, INIT_H_POS, INIT_V_POS, PREV_H_POS, L, R, reset_road, and size. The outputs of the module are road and road_h_address. The road output defines the area of a road segment. The road_h_address gives the current horizontal addresses with respect to the center of the road segment. This output is used to determine the next horizontal address for the following road segment to be loaded to the top of the screen. In order to keep track of the seven road segment position we use 16-bit counters. We will need two separate counters for the vertical position and the horizontal position.

Road-h-counter: The inputs to this module are: clock, L, R, reset, rollover, INIT_H_POS, and RESET_H_POS. The output to this module is road_h_address. This dictates the horizontal position of a road segment. To keep track of this we call an instance of the 16-bit counter from Lab 4. The input INIT_H_POS is the initial horizontal value of the road segment seen at the start of the game. There's also a RESET_H_POS meant to give the new position of the road segment as it generates from the top of the screen. This value is determined by the horizontal address of the preceding road segment summed with a random offset. These two 16-bit values are sent to a mux to determine which of the two should be loaded into the 16-bit counter horizontal road address counter. The RESET_H_POS bus is put in the most significant mux input as the select input is determined by the rollover signal. This signal tells us when the vertical address counter is rolling over, which means we want to generate a new horizontal address. The least significant mux port is populated by the 16-bit INIT_H_POS input. This bus loads the initial hard-coded value of the Rainbow Road. The two load signals for the 16-bit counter instance within our module is the OR of inputs reset and rollover. We want to load the value from our mux whenever either of these signals is high. The mux determines the proper D input according to the load signal. The L and R input signals tell our car to steer right or left. These ports are populated by the respective L and R signals from genRoad module. To give the illusion of steering the static car we increment and decrement the road-h-address to make it seem as the car is moving left and right along the road. The L input is sent to the down port of our 16-bit counter instance, while the R input is sent to the up port of the 16-bit counter. The only other functionality to employ in this module is to ensure the horizontal motion of the road is at a speed of two pixels per frame. In order to accomplish this I shifted the bits of my D port input right prior to sending it to the 16-bit counter module. This is necessary because the method I employ to move twice a frame depends on the fact that shifting bits to the left can multiply them by two. Due to this operation after counting we can effectively count up and down twice per frame. The only output to this module is the road-h-address. Send this to the CreateObj module to assemble the road boundaries.

After completing the road-h-counter we move on to create the road-v-counter.

Road-v-counter: The road-v-counter has four inputs: clock, up, reset, and INIT_V_POS. The outputs are the road-v-address and the rollover signal to signal the road-h-counter to reset with a new horizontal position. Within this module we call a 16-bit counter to keep track of the vertical address. We connect the clock to the clock as always. The up port of the genRoad module is connected directly to the input port of the road-v-counter. The down port of the instance is grounded since we don't ever want to count down. The load value for the D port of this instance is chosen by a mux. We load a 16-bit bus of zeroes or the INIT_V_POS depending on the value of the select. The select signal is the rollover signal which tells the road-v-counter to reset to the top of the screen. Therefore the most significant port of our mux is populated by the 16-bit zero value. The other port is given the INIT_V_POS bus because we want to load this value whenever we aren't rolling over. The load input of the 16-but counter instance is then dictated by the OR of the signals rollover and reset. The rollover signal is high when the road-v-address is greater than 557 and the up signal is high. This signal is one of the outputs of the module. The UTC and DTC outputs of the 16-bit instance are unnecessary in this design so we assign them to local wires within the module and forget about them. The final output is the road-v-address. This tells us the vertical position of the road. This output is sent to the genRoad modules along with the road-h-address to create the final boundaries of the road. The vertical motion of our road segments should be two pixels per frame like the horizontal motion. To achieve this we shift the bits to the left as they exit the 16-bit counter, but for this to work we halve the D input values to ensure we stay don't accidentally spawn a segment off screen.

After defining the road-h-counter and road-v-counter return to complete the genRoad module.

We call an instance of the road-h-counter and road-v-counter to use for creating our road segment. The clk input port of the road-h-counter is connected to the clk input port of the genRoad module. The L input port is fed the L input of the genRoad module AND'ed with the frame and drive inputs to ensure we are changing the horizontal address of the road segment once a frame while in the DRIVE state. The same is done with the R input of genRoad to the R input of the road-h-counter. The reset port of our road-h-counter is fed the reset_road signal from the genRoad input port. The INIT_H_POS input of the genRoad module is fed directly to the INIT_H_POS port of the road-h-counter module. The last input port of the road-h-counter is the RESET_H_POS. This 16-bit input is determined by the sum of a random offset and the PREV_H_POS input from the genRoad module. To implement this functionality, call an instance of the RandomLFSR from Lab 5. This module outputs an 8-bit random value. For our case since we only want a select set of random values, all multiples of 8. To achieve this I stored the output of this module into an 8-bit wire. I then used the bits [6:3] of the random output to determine the random offset. The bit random[6] is used to determine whether the next offset is a sum of the previous address or a subtraction. This functionality is implemented using a mux. The two inputs to the mux are the two different sim and differences of the values represented by the unused bits(rand[5:3]) with the PREV_H_POS. The least significant input of the mux is the sum of the offset with the previous horizontal road segment address. The more significant input is the difference of the two. The output of this mux is then sent to the RESET_H_POS wire and fed into the RESET_H_POS input port of the road-h-counter. The final port of our road-h-counter is the output road-h-address and it is output to the higher-level module CreateObj. After acquiring the road-h-address from this module we can begin declaring the region of the road. The region to declare is the horizontal one. In order to do this according to the lab manual we need to create different horizontal offsets to increase the size of road_x(width of the road). There are eight different offsets calculated with the formula given in the lab manual(8+16i(i=value represented by switches four through six(sw[6:4]=size[2:0]))). To implement this function according to the combination of switch values, we can use an 8-to-1 mux. The eight inputs would be given according to the above formula, while the select would be determined by the size[2:0] bus, which is connected to our switches(sw[6:4]). After this we assign the road_x range with respect to the value of the road-h-address. To do this we need to add and subtract half of the offset from the mux output to the current road-h-address. The lower bound is the difference while the upper bound is the sum. We use comparison operators to define the road_x signal(road_x = (H_address<(road_h_address+(H_OFFSET>>1))&&(H_address>=(road_h_address-(OFFSET>>1)))). After declaring the horizontal boundaries of the road segment we can move forward to define the vertical boundaries of the road segment. To do this we call an instance of our road-v-counter module. This module has inputs clk, up, reset, and INIT_V_POS. The outputs of the module are road_v_address and rollover. The up port of this instance is connected to the input of the genRoad module, frame, anded with the another input, drive. The reset port is connected directly to the input of genRoad, reset_road. The clk input port is also directly connected to the clk input port of the road-v-counter module. The INIT_V_POS input has the input of the genRoad module INIT_V_POS connected to it. The rollover output of this instance of the road-v-counter is input to the road-h-counter. The final output, the road-v-address is used to define the vertical boundaries of the road segment. The height of the road segments is static throughout their transition from the top to the bottom of the screen. Therefore, there isn't much to alter for this module to function properly. The only thing to modify are the vertical boundaries of the road segment as the road segment needs to appear from the top of the screen gradually as a scroll rather than appearing in an instant. To implement this function, we add a mux to pick between two outputs for the range of the vertical boundaries. To do this we use a select that tells us when the road-v-address is less than 80 pixels. The two inputs to the mux output the range of the road segment dependent on its vertical position. When the address is greater than 80 we want to output the road segment with both upper and lower boundaries. When the address is less than 80 we want to output the road segment only with the upper boundary, as if we include the bottom boundary, the condition would not be met for populating the screen until the full segment vertical range would be on the screen. The output of this mux is the road_y range(height of the road segment).

After acquiring the road_y boundary and the road_x boundary we can define the whole road segment within genRoad. To do this we simply AND the two ranges road_x and road_y. This concludes the genRoad module design.

Returning up a level to the CreateObj module, we can create seven road segments for our design by calling seven instances of the genRoad module. So we call seven instances of the genRoad module. The INIT_V_POS input for the seven roads are separated by 80 pixels each. The v-address and h-address inputs come from the VGAcontroller. The clk input comes from the lab7clks module instance at the top level. The drive input comes from out state machine. The INIT_H_POS input was chosen arbitrarily with the restriction of keeping every segment less that 64 pixels away. The frame input is the edge detected vsync signal defined in the top module. The PREV_H_POS input comes from the previous road segment to have been generated. The outputs of the road-h-address are connected in a ring from least significant road to most significant. This ensures the horizontal address of a newly rolled over road segment is within a reasonable distance from the last. The L and R inputs come from the btnL and btnR inputs from the top module after they have been synchronized with the system. The reset_road port is connected to the output of the state machine which tells us when we want to load all initial values. The final output, size come from the switches four to six. These switches dictate the width of the road. This leads to seven road segments scrolling from the top of the screen to the bottom and reloading at the top with a random offset.

An additional function to include now in the CreateObj module is the flashing of the car object. To get this functionality we use the flash and flash_car inputs. Within this module we previously defined the static_car range. We can create a flashing car if we use the flash input and AND it with the static_car definition. We then feed both inputs to a mux to determine the true output of our car object. The select for this mux is the flash_car signal. This signal comes from the state machine. Afterwards we have one output left to define for this module. The final output to define for this module is the off_road signal. This off_road signal is dependent on the current position of the car and whether it is on a road segment or not. The road output of this CreateObj module is a 7-bit bus and defines the seven different road segments that populate our screen at a time. Using this bus we can define the off_road signal. To check if the car was still on the road I simply did an OR of the 7-bit road bus to ensure the road segment objects were connected. I then AND'ed the output of this bus to the static_car area range. This tells us when the car is on the road but only when the pixel address of our counters is within the static_car range. If we fail to do this we get an invalid off_road signal. To employ this functionality we use a mux to pick whether to output the logic for the off_road signal, or a 0 value to ensure the game doesn't end before the off_road signal has a chance to check whether or not the road is on any of the road segments. This output is then fed to an flip-flop which is reset once a frame. This concludes the design of the CreateObj module. The off_road signal is sent to our state machine to tell us when we want our GAME_OVER sequence to begin.

After completing this module we can move on to building a time counter to keep track of time for our state machine.

<u>Time Counter</u>: The time counter is used to keep track of 2_seconds, the flash output, the half_second output, and the q_sec output. These outputs help us time our system appropriately. The time counter only needs a clk input, a ce, and a reset. The clk is given by the lab7clks module. The ce is connected to our frame signal, because we know our VGAcontroller iterates through its pixel addresses at 60 frames a second. Therefore, if we round up to 64, we can calculate seconds according to the bit significance of this counter output.

<u>LED countdown</u>: the LED_countdown module is a 16-bit shift register that shifts a bit into the LFSR once a half second. The inputs for this module are: in, ce, reset, and clk. The clk is connected from the top wire. The ce is connected to the output of the state machine, countdown. The in input is given by the half_sec signal. The reset signal is given by the setup signal. The output of this module is a 16-bit bus but I only use the 4 least significant bits to create a new bus with the correct light-up sequence detailed in the lab manual(countdown[0],3'b0,countdown[1],3'b0,countdown[2],3'b0,countdown[3],3'b0).

This bus animating the LED_countdown is sent to a mux because there are two different LED properties.

<u>LED animation</u>: The other LED property is the LED_animation giving the DRIVE state animation. This animation is simply a collection of three ring counters revolving at a rate of once a qsec. The 16 LED_animation bus is divided into four sections and each section is given a 4-bit bus output of the ring_counter. This 16-bit bus is sent to the mux that was previously mentioned to determine whether to output the LED_countdown or LED_animation bus to the LED's.

The select for the mux is the drive signal. This means we connect the LED_animation bus to the more significant input port of our muc, while the least significant port is populated by the LED_countdown bus. This ensures we get the proper output for our LED's. But to further ensure the LED's are only on when we want them to be we cascade another mux to determine whether we want to set the LED's or if we want them off. The next module to complete is the score counter.

<u>Score Counter</u>: The score counter is a 16-bit counter instance with an enable that increments every qsec. This enable is AND'ed with the drive signal from our state machine as well as the UTC of the instance, because we don't want our score to increment past the max value. The down port of the counter is grounded, the clk is connected from the top wire, the ld port is connected to the setup signal from the state machine. The D input is a bus of 16 zeroes. The UTC and DTC outputs are output to wires.

The Q output is set to the score wire, which is to be sent to be converted to populate the 7-segment LED.

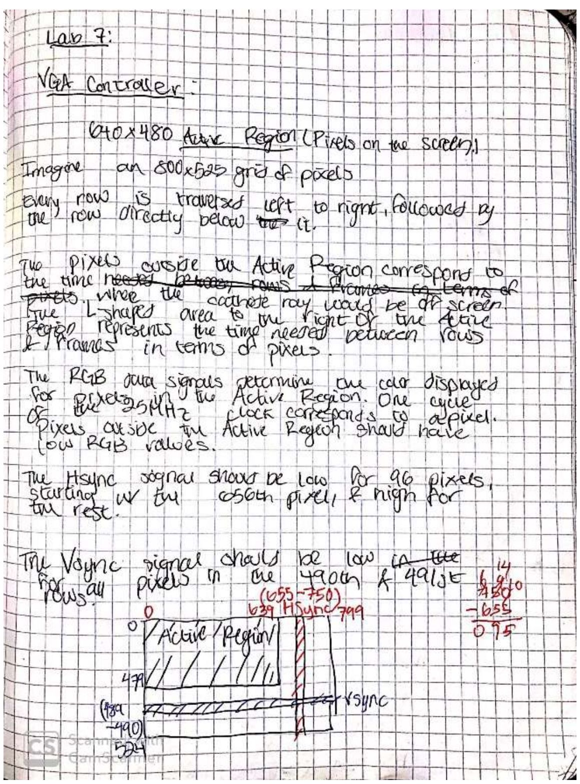
After finishing this design we can move on to the Colorize module, which gives the outputs of the vgaRed, vgaBlue, and vgaGreen outputs. I didn't use any sophisticated method to define the colors of my objects. I simply listed out the value of all the color outputs for every object in a list. I then wrote the associated bit values to the right of the list of objects. This allowed me to see the different vga output combinations for all the objects. I then simply ensured that all the color signals were on when I wanted them to be. I OR'ed all the objects that needed the same color value. The table I made made it easier to see when the values of every vga color were given by the columns of the table.

This concludes every new design required to get this lab to function. The rest of the modules used in the top level are defined in previous labs. We simply connect the modules we've created to the appropriately labeled ports in the top module. After connecting all these wires the project is complete.

Conclusion: All in all this lab was very time consuming. The most difficult part of this lab was getting started. There were so many things to get done I wasn't really sure what to do first. Another difficult part of this lab for me was trying to debug by myself. I'm glad to have been checked off in the end though.

Appendix:

Lab Notes:



Lab 7:

VGA Controller:

640x480 Active Region (Pixels on the screen)

Imagine an 800x525 grid of pixels

Every row is traversed left to right, followed by the row directly below ~~the~~ it.

The pixels outside the Active Region correspond to the time ~~needed between rows & frames in terms of pixels~~ where the cathode ray would be off screen. The L-shaped area to the right of the Active Region represents the time needed between rows & frames in terms of pixels.

The RGB data signals determine the color displayed for pixels in the Active Region. One cycle of the 25MHz clock corresponds to a pixel. Pixels outside the Active Region should have low RGB values.

The Hsync signal should be low for 96 pixels, starting w/ the 656th pixel, & high for the rest.

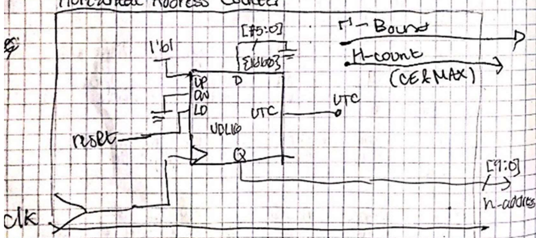
The Vsync signal should be low ~~in the~~ for all pixels in the 490th & 491st rows.

VGA Controller:

Horizontal Address Counter)

- inputs:
  - clk
  - CE
- outputs
  - hsync (Low from 655-750)
  - h-address / current horizontal pixel address.
  - UTC

Horizontal Address Counter



Want our counter to roll over @ value 799 & CE

assign h_MAX = 10'd799 ;

assign reset = (~|(MAX ^ Q))&CE ; (Q: n-address)

reset when we are @ max value (799) & CE

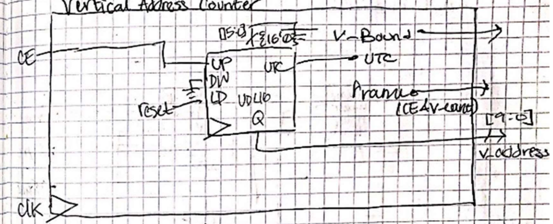
assign HSync = ~((Q ≥ 655) | (Q ≤ 750)) ;

Vertical Address Counter:

inputs:
- CLK
- CE

outputs
- VSync (low from
- V_address(
- UTC / frame signal

Vertical Address Counter



assign V_MAX = 10'd524;
//max value for our vertical address counter

assign reset = ((~|(MAX ^ v_address)) & CE);
//reset tells us when to roll over
//back to zero from MAX value

assign VSync = ~((v-address ≥ 489) | (v-address ≤ 490));

assign v-count = CE & MAX;

VGA Controller Output:

assign ActiveRegion = (Horizontal Boundary) | (Vertical Bound));

assign wire Hor_Bound = (h-address ≤ 639); ← Output H_count

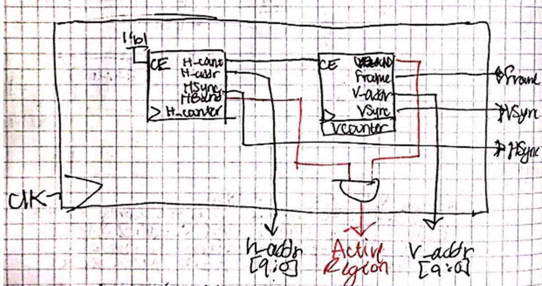
assign wire Ver_Bound = (V-address ≤ 479); ← Output V_count

~~assign~~

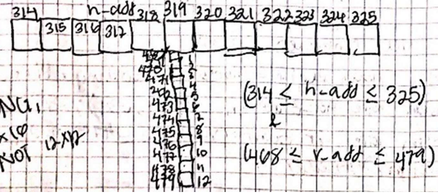
VGA Controller:

inputs:
- clk

outputs:
- hsync
- VSync
- h_address [9:0]
- V_address [9:0]
- ActiveRegion



CAR Dimensions (According to h-address)



(314 ≤ h-add ≤ 325)
&
(468 ≤ v-add ≤ 479)

WRONG,
16x16
NOT 12x12

Color Module:

Determines what color to output onto VGA

inputs: (*This list will increase depending on necessary inputs from State Machine).
- V_address
- h_address

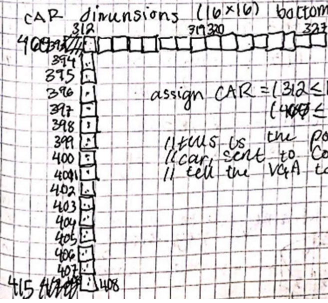
outputs
- red [3:0]
- Blue [3:0]
- Green [3:0]

assign Blue = {ActiveRegion, ActiveRegion, Active Region}

Objects Module: Determines the ranges for the shapes of the objects we want to populate our screen with.

~~assign CAR = (314 ≤ h_addr ≤ 325) & (468 ≤ v-addr ≤ 479);~~

CAR dimensions (16×16) bottom center of screen



assign CAR = (312 ≤ h-addr ≤ 327) &
(400 ≤ v_addr ≤ 415);

//this is the position of the
//car sent to Color Module to
//tell the VGA to output white

Each road segment is 80 pixels tall
∴ we need @ least 7 road segments.

Use top 4 bits of 7-bit LFSR

0 0 0 0 0 0 0
64 32 16 8 4 2 1

0000 → 0 → 0/8 = 0
0001 → 8 → 8/8 = 1
0010 → 16 → 16/8 = 2
0100 → 32 → 32/8 = 4
0101 → 40 → 40/8 = 5
0110 → 48 → 48/8 = 6
0111 → 56 → 48/8 = 7
1000 → 64 → 64/8 = 8
1001 → 72 → 72/8 = 9
1010 → 80 → 80/8 = 10
1011 → 88 → 88/8 = 11
1100 → 96 → 96/8 = 12
1101 → 104 → 104/8 = 13
1110 → 112 → 112/8 = 14
0011 → 24 → 24/8 = 3
1111 → 120 → 120/8 = 15

14 × 8

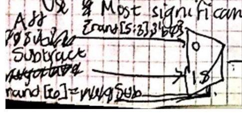
120

56 × 2 = 112

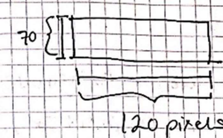
104 / 8 = 13

0000 000 {3'b0}

Use ~~4~~ Most significant bit to dictate ~~positive~~ Addition vs. ~~negative~~ Subtraction.



Road Segment Counters



Vertical Address Counter:
Counts UP & resets when we reach V-address 480

The horizontal address is hard-coded initially.

We want to load ~~a random offset~~ the horizontal address w/ the sum of difference of our random offset w/ the road segment directly below it

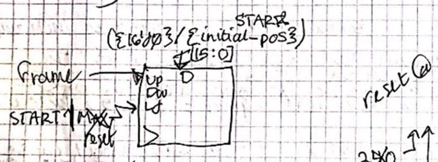
Our Left & Right buttons should cause all road segments to shift left or right. This is equivalent to adding or subtracting / counting up or down the horizontal address counters.

7 Road segment counters
7 vertical
7 horizontal

70 pixels tall
120 pixels wide (for now) - (Add variability after)

Needs a CE (Does not count in START or LOSE/GAME_OVER

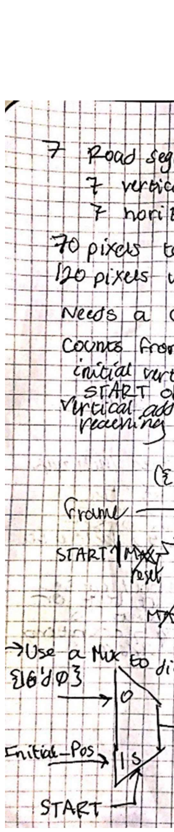
Counts from initial position.
initial vertical position is loaded @ START of game. Afterwards the vertical address is reset to Ø after reaching V-address 480.



~~MAX~~ = 10'd480; ~(Q ^ 10'd ~~480~~ 290)

reset = ~|(Q ^ 10'd ~~480~~ 290) & Up;

→ Use a Mux to dictate whether to load value from the START of game or ~~random value~~ back to V-addr Ø.



Ld_ROLL;
Ld_INIT;

We want our road segments to move 2 pixels per frame, how? (vertically)

Use Frame as a CE for the counter, but shift the output left (<<) by one bit to Multiply by 2. Effectively moving the road segment 2 pixels per frame.

→ VPOS_INIT = ~~280~~ Ø, 70, 140, 210, 280, 350, 420

Horizontal Road Segment Counter:
Hard-coded initial values

→ After rolling over once the output values become offset by a random value from the LFSR randomizer plus the h_address of the ~~current~~ road segment under it.

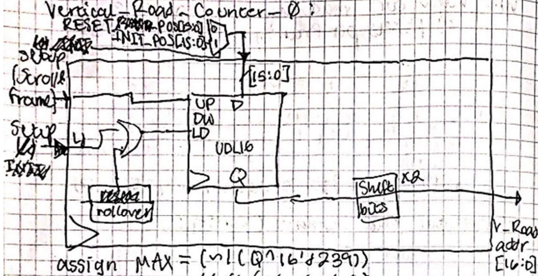
→ The horizontal position should only be updated when rolling over & when we get the signal from btn L or btn R.

∴ We want two different load signals & two different load values.

ld_RAND // signals our counter to load the new position
ld_INIT // signals our counter to load the initial position (Hard-Coded)

rand_in[15:0]; // random offset to load
INIT_in[15:0]; // Initial horizontal position

Vertical_Road_Counter_Ø:



assign MAX = (~|(Q^16'd239))

~~assign reset = (~|(Q^16'd239))~~

assign rollover = MAX&CE;

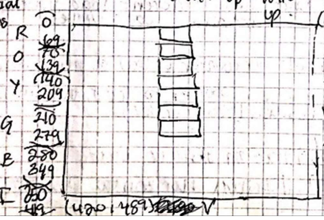
(INDEX VERT POS COUNTERS BY

~~assign~~ setup ⇒ input that comes from State Machine

//tells our counter to Load setup

→ Don't want to count up while we are setting up.

Initial Positions



(0, 69) (69, 138) (138, 207)

(207, 276) (276, 345)

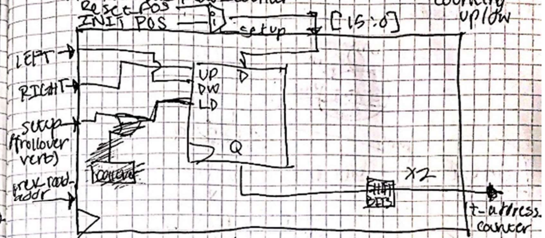
(345, 414) (414, 483)

V_box_Ø =

{10'd0, V_address}

*Shift outputs to the left when counting up/dw

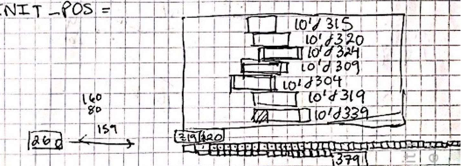
Horizontal_Road_Counter



assign MAX = (~|(Q^16'd319)); //assign MAX value

assign rollover = MAX&CE; //assign rollover signal

INIT_POS =



INIT_POS_1 = 10'd319; //Initial position of our horizontal Blocks

INIT_POS_Ø = 10'd339

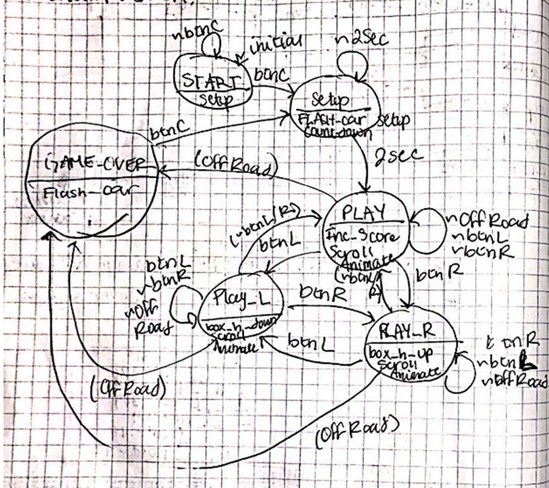
INIT_POS_2 = 10'd304;

INIT_POS_3 = 10'

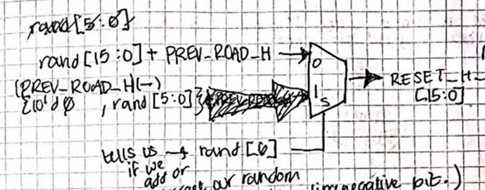
RESET_POS_1 = (road_H_address[15:0]+RAND_OFF) ~~(RESET_POS_0+RAND_OFF)~~;

RESET_POS_Ø = (RESET_POS_6+RAND_OFF);

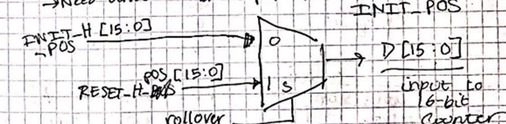
RainbowRoad_SM:



mx2_1x16: Need one to pick between Adding & Subtracting the random offset to the preceding roadsegment to load @ top of screen.



→ Need another Mux to pick b/w RESET_H_POS & INIT_POS

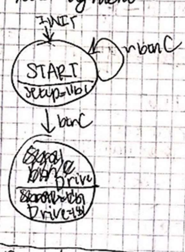


→ Shift Right >>1)

*(B/c we shift Left on the way out to move 2 pixels @ a time.)

Buttons L & R working right on H_counter.
Need to fix Random Offset for rolling over ~~&~~ (Horizontal Address).

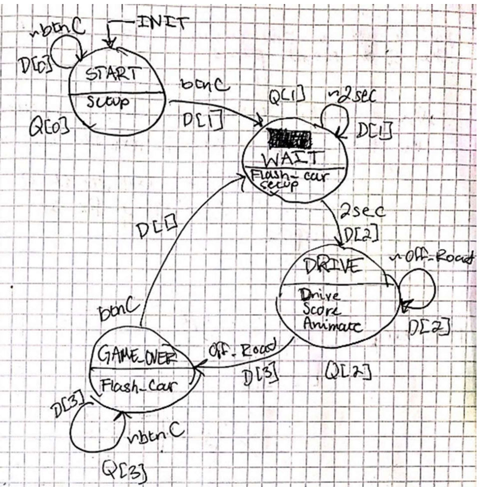
→ Need a Simple State Machine to test several road segments.



PB	NS		Output	
	btnC=0	btnC=1	Setup	Drive
START	START	DRIVE	1	0
DRIVE	DRIVE	DRIVE	0	1

$Q_1 Q_0$
0 1
1 0

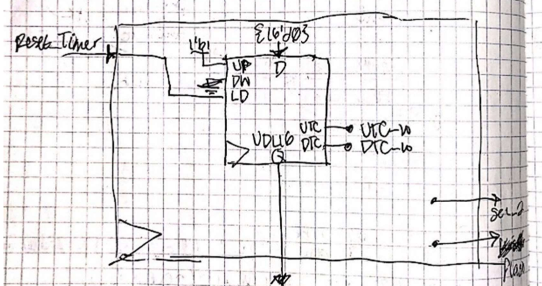
D[0] = (Q[0] & (~btnC));
D[1] = (Q[0] & btnC) | (Q[1]);



D[0] = Q[0] & (~btnC);
D[1] = (Q[0] & btnC) | (Q[1] & (~sec_2)) | (Q[3] & btnC);
D[2] = (Q[1] & sec_2) | (Q[2] & (~Off_road));
D[3] = (Q[2] & Off_road) | (Q[3] & (~btnC));
assign Flash_car = Q[3] | Q[1];
assign Drive/Score/Animate = Q[2];
assign setup = Q[0] | Q[1];
assign Reset_Timer = (Q[0] & btnC);

Need to add a time-counter to keep track of the # of seconds passed.

→ Use a 16-bit counter, or maybe smaller. The ~~frame signal~~ screen is refreshed 60 times a second, but we can consider 64 a second, since it's close enough for our purpose.



time_bits[15:0]
time_bits[5] = half_sec → used to blink car
time_bits[6] = ~~second~~ sec
time_bits[7] = 2_sec

→ to Flash car

$$RGB = \underbrace{(car \& (\sim Flash\_car))}_{\text{Static car}} \mid \underbrace{(car \& (Flash\_car) \& (Flash))}_{\text{Flashing car}};$$

R { (4'b{ (car & (~Flash_car)) | (car & Flash_car & (Flash));
G {
B {

H_boundary variability

→ make the ~~~~
horizontal boundary input for road generation ~~~~ to define the different combinations w/ the given switches.

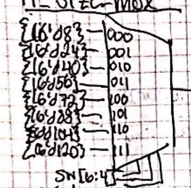
SW [6:4]

	SW[6:4]		Offset
0	000	8+0	8
1	001	8+16	24
2	010	8+32	40
3	011	8+48	56
4	100	8+	72
5	101		88
6	110		104
7	111		120

~~054~~

→ We can get this to work using a mux.

H_size-mux



* Add this module to gen_road module.
→ Add m8_1 module

→ After, figure out the RAINBOW setup

R = {(road_6) | (...), (road_6) | (...), (road_6) | (...), (road_6)}

→ R = {(4'{road_6 & ActiveRegion}) | (4'{car & ActiveRegion})}

| 4'{road_5 & ActiveRegion}}

G = {{1'b0}, (3'{road_7 & ActiveRegion}) | ({car & ActiveRegion})

| {4'{road_5 & ActiveRegion}

off_road = ~((car & road_0) | (car & road_1) | (car & road_2) | (car & road_3) | (car & road_4) | (car & road_5) | (car & road_6));

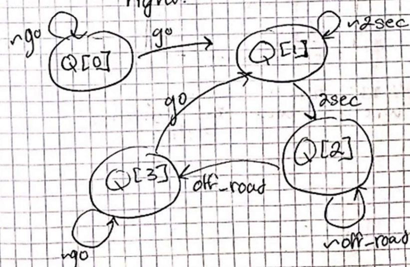
a & b + a & c + a & d
= a & (b + c + d)

→

off_road = ~(car & (|road[6:0]));

→ 7-segment display shows the current score

→ State Machine Not transitioning right.



Simulating our state machine shows the transitions are working properly.

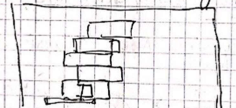
When the board is programmed though, State Q[2] is skipped over in less than a second.

In Q[2] we are supposed to drive, which causes the road to start scrolling.

The state should remain until we receive the off_road signal.

The off_road signal is an output of ~~gen~~ Create Obj.

It is the AND of our (car & (|road)); car & the OR of the road segments.



I think there's something wrong with the off_road signal

→ To try and debug, I've output the off_road signal to LED[4]

the complement is output to LED[5].

I'm going to simulate the create_obj Module to see whether or not the road[6:0] segments are defined properly

→ off_road appears high @ all times when simulating top_module. Not sure why. Not sure how long I should run the simulation for either. For some reason my input btnC wouldn't change in the waveview even though I change it in my simulation source.

→ Wait in lab Monday/Tuesday to see if anyone can help me figure out what this bug comes from.

→ Until then, assign proper colors for Rainbow Road.

Assigning colors within the color module

road			R	G	B	binary
[0]	R	0x	F	Ø	Ø	0b-1111-0000-0000
[1]	O	0x	F	7	Ø	0b-1111-0111-0000
[2]	Y	0x	F	F	Ø	0b-1111-1111-0000
[3]	G	0x	Ø	F	Ø	0b-0000-1111-0000
[4]	B	0x	Ø	Ø	F	0b-0000-0000-1111
[5]	I	0x	2	2	5	0b-0010-0010-0101
[6]	V	0x	8	Ø	F	0b-1000-0000-1111

R G B
[3:0] [3:0] [3:0]

~~vga Red~~

vga Red[0] = (|road[2:0]);
vga Red[1] = (|road[2:0])|(road[5]);
vga Red[2] = (|road[2:0]);
vga Red[3] = (|road[2:0])|(road[6]);

vga Green[0] = (|road[3:1]);
vga Green[1] = (|road[3:1])|(road[5]);
vga Green[2] = (|road[3:1]);
vga Green[3] = (|road[3:2]);

vga Blue[0] = (|road[6:4]);
vga Blue[1] = (road[6]|road[4]);
vga Blue[2] = (|road[6:4]);
vga Blue[3] = (road[6]|road[4]);

road_x = (h_address ≥ H_ADD)|(h_address < H_ADD+OF

Score_Counter

→ Counts up 4 times a frame

→ 4 times a second

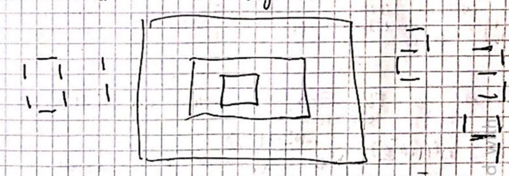
→ Simply use 16-bit counter w/ UP enable input = (DRIVE&qsec);

off_road

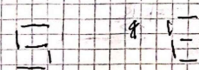
V_coordinate &

H_coordinate

need to be within the car walls to know if we want to check for our car overlapping the road segment



(H_address ≤ static_car) & (H_address ≥



→ Problem w/ Score_Counter

The score_counter is counting up too fast.

Why is it counting so fast?

→ Check Enables & ports.

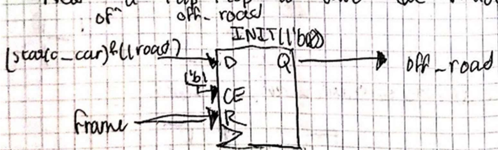
→ Make sure everything is connected

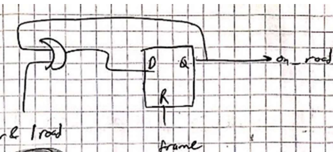
→ For our off_road signal we should only be checking for whether the car is on the road, if the H_address and V_address of our screen pixels are within the car's defined X & y coordinates, ~~then~~.

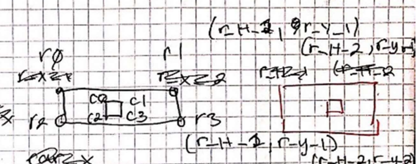
These coordinates are hard-valued so they shouldn't be hard to implement.

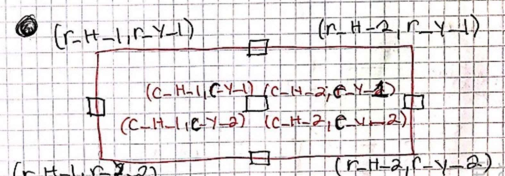
Whenever the H-counter & V-counter are within these boundaries we want to test whether or not the car is on the road.

→ Need a Flip-Flop to save the validity of off_road









(c_H_1 < r_H_2)
(c_H_2 > r_H_1)
(

LED Animations:

1) counting

halfsec
+ state
condition
output

LEDshift
Lab5

4

{LED[3], LED[7], LED[11], LED...

2) running

input
halfsec & state and then

4

Led[15:12]

x4

4

Led[11:8]

countdown_bits[0] = LED[15];

countdown_bits[1] = LED[11];

countdown_bits[2] = LED[7];

countdown_bits[3] = LED[3];

score counter is broken again

→ counts all the way up to ffff in less than a second.

→ ~~We should advance the score four times a second, (once every qsec)~~.

We want to advance the countdown bits every half second when we are in the WAIT state.

(halfsec & drive) → Adv

an [3:0] — LED[15:12]

clk → r_c

" " LED[11:8]

" " LED[7:4]

" " LED[3:0]

md_1x4

Lab7 checkoff 60%

3/10/20
4:25pm
6898
RH

Source and Schematic

```
`timescale 1ns / 1ps
////////////////////////////////
////////////////////////////////
/////////////////
// Company:
// Engineer:
//
// Create Date: 03/06/2020
12:05:35 AM
// Design Name:
// Module Name: Lab7_top
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////
/////////////////////////////////
////////////////


module Lab7_top(
    input clkin,
    input sw,
    input btnL,
    input btnR,
    input btnC,
    input [2:0]size,
    output hsync,
    output vsync,
    output [3:0] vgaRed,
    output [3:0] vgaBlue,
```

```
    output [3:0] vgaGreen,
    output [15:0] led,
    output [6:0] seg,
    output [3:0] an
    //output car_off_road: output
for simulation
    );

    //declare wires for lab7_clks
outputs
    wire clk, digsel;

    //call instance of lab7_clks
    lab7_clks not_so_slow (
        .clkin(clkin),
        .greset(sw),
        .clk(clk),
        .digsel(digsel)
    );
```

```
    //declare wires for
time_counter
    wire sec_2, flash,
ResetTimer, frame,qsec;

    //call instance of time
counter
    time_counter count_time(
        .clk(clk),
        .ce(frame),
        .r(ResetTimer),
        .sec_2(sec_2),
        .flash(flash)
    );



    //declare wires for qsec and
```

```
halfsec counters
    wire [15:0]time_bits;
    wire halfsec;

    //call counter to keep track
of qsec and halfsec
    UDL16_count count_timer(
        .clk(clk),
        .up(frame),
        .dw(1'b0),
        .ld(time_bits==16'd30),
        .D({16{1'b0}}),
        .UTC(),
        .DTC(),
        .Q(time_bits)
     );

    assign qsec =
(time_bits==16'd15) |
```

```
(time_bits==16'd30);
    assign halfsec =
(time_bits==16'd30);



    //wires for score counter
output
    wire score_UTC, score_DTC;
    wire [15:0]score;

    //declare wires for state
machine
    wire setup, drive, go,
flash_car, off_road, on_road,
ResetScore,countdown;

    //declare wire to store SR
countdown bits
    wire [15:0]countdown_bits;
```

```
    //call ring counter to
control the bits for our LED
countdown
    SR_16b LED_countdown(
        .clk(clk),
        .in(halfsec&countdown),
        .ce(1'b1),
        .reset(~countdown),
        .Q(countdown_bits)
    );

    //assign countdown_led wire
    wire [15:0]countdown_led;

    assign countdown_led = {
        countdown_bits[0],3'b0,
        countdown_bits[1],3'b0,
        countdown_bits[2],3'b0,
```

```
        countdown_bits[3],3'b0
    };


    //implement LED animation
    wire [15:0]animation_led;

    //call four instances of ring
counter
    ring_counter animate_3_0(
        .Advance(halfsec&drive),
        .clk(clk),
        .an(animation_led[3:0])
    );
    ring_counter animate_7_4(
        .Advance(halfsec&drive),
        .clk(clk),
        .an(animation_led[7:4])
    );
```

```
    ring_counter animate_11_8(
        .Advance(halfsec&drive),
        .clk(clk),
        .an(animation_led[11:8])
    );
    ring_counter animate_15_12(
        .Advance(halfsec&drive),
        .clk(clk),
        .an(animation_led[15:12])
    );


   //declare wire to pick between
animate and countdown bits
   wire [15:0]set_led;

   //call 2_1x16 mux to determine
proper led output
   m2_1x16 pick_LED(
```

```
        .in0(countdown_led),
        .in1(animation_led),
        .sel(drive),
        .o(set_led)
   );

   //m2_1x16 mux to determine
proper output led
   m2_1x16 final_LED(
        .in0({16{1'b0}}),
        .in1(set_led),
        .sel(drive|countdown),
        .o(led)
   );

    //call score counter
    UDL16_count count_score(

.up(qsec&drive&(~(score_UTC))),
```

```
        .dw(1'b0),
        .ld(setup),
        .clk(clk),
        .D(16'd0),
        .UTC(score_UTC),
        .DTC(score_DTC),
        .Q(score)
    );


    //declare wire to store
dig_sel bits
    wire [3:0] dig_sel;

    //declare wire to store hex
value for conversion
    wire [3:0] H;

    //output score to 7-segment
```

```
display
    //call ring counter to
generate dig_sel bits
    ring_counter
select_dig(.clk(clk),.Advance(dig
sel),.an(dig_sel));

    //call selector to choose hex
value to convert to seg
representation
    selector
select_hex(.sel(dig_sel),.N(score
),.H(H));

    //send Hex value be converted
to seg representation
    hex7seg
convert_hex(.n(H),.seg(seg));
```

```
    //assign an output

    //declare wires to hold
different an values
    wire [3:0]static_an;
    wire [3:0]flash_an;

    assign an =
{~dig_sel[3],~dig_sel[2],~dig_sel
[1],~dig_sel[0]};



    //edge detector to tell us
when to start a game
    edge_detector
detect_btnC(.btnIN(btnC),.clk(clk
),.o(go));
```

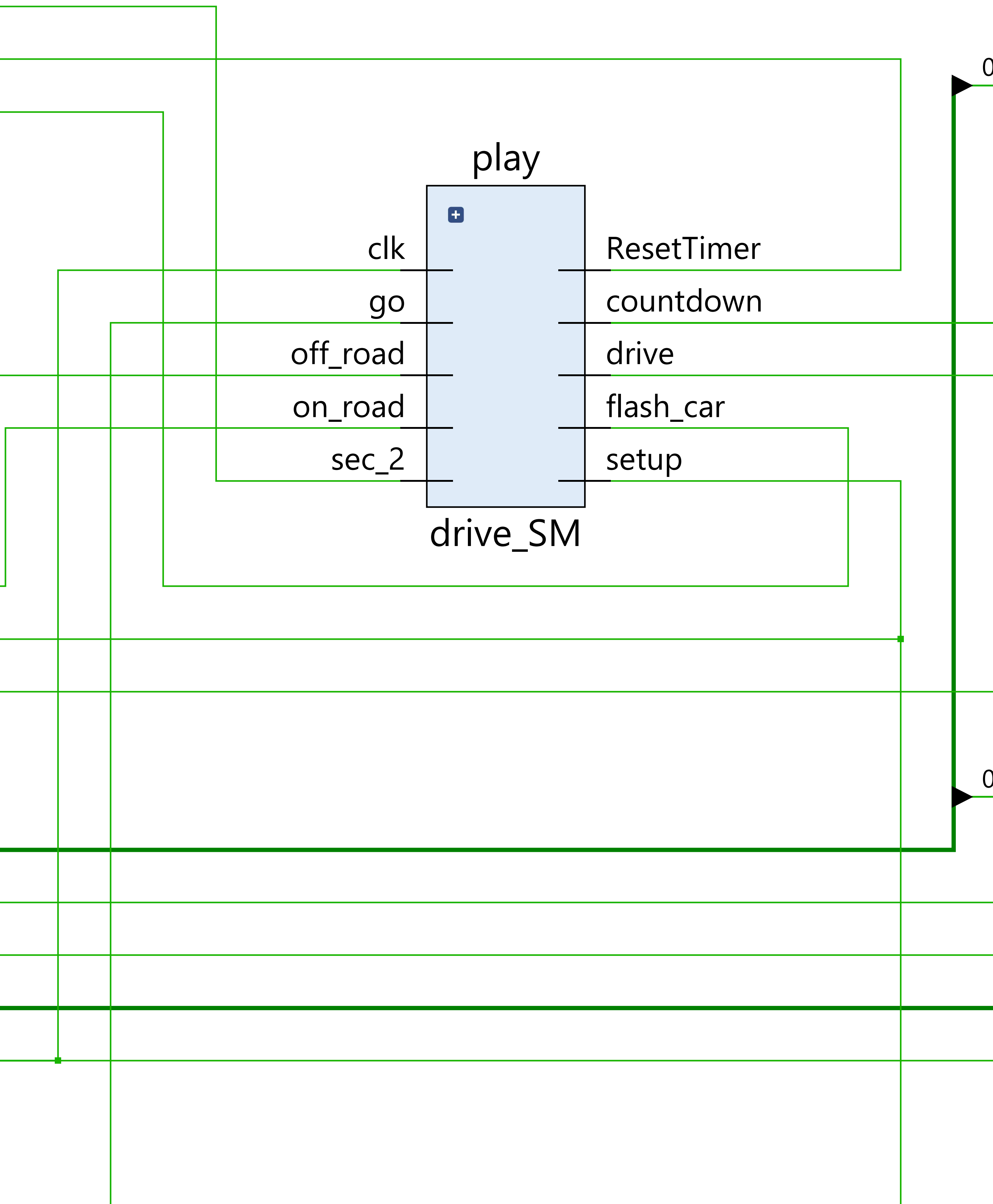
```
    //call instance of state
machine
    drive_SM play(
        .clk(clk),
        .go(go),
        .sec_2(sec_2),
        .off_road(off_road),
        .on_road(on_road),
        .drive(drive),
        .setup(setup),
        .flash_car(flash_car),
        .ResetTimer(ResetTimer),
        .countdown(countdown)
        //.Q(led[3:0])
    );



    //declare wires for
```

```
VGAcontroller output
    wire [15:0]H_address;
    wire [15:0]V_address;

    //call instance of
VGAcontroller
    VGAcontroller control(
        .clk(clk),
        .hsync(hsync),
        .vsync(vsync),
        .H_address(H_address),
        .V_address(V_address)
    );


    //call instance of edge
detector to give a frame signal
    edge_detector
detect_frame(.btnIN(vsync),.clk(c
```

```
lk),.o(frame));

    //declare wire for Create_Obj
output
    wire ActiveRegion, car,L,R;
    wire [6:0]road;

    //instantiate FF's to
synchronize button signals
    FDRE #(.INIT(1'b0) ) btnL_FF
(.C(clk), .R(1'b0), .CE(1'b1),
.D(btnR&(~btnL)), .Q(L));
    FDRE #(.INIT(1'b0) ) btnR_FF
(.C(clk), .R(1'b0), .CE(1'b1),
.D(btnL&(~btnR)), .Q(R));

    //call instance of Create_Obj
    Create_Obj create(
        .clk(clk),
```

```
        .R(R),
        .L(L),
        .frame(frame),
        .setup(setup),
        .drive(drive),
        .flash(flash),
        .flash_car(flash_car),
        .size(size),
        .H_address(H_address),
        .V_address(V_address),

.ActiveRegion(ActiveRegion),
        .car(car),
        .road(road),
        .off_road(off_road),//off
road and on road should only be
on at drive
        .on_road(on_road)
    );
```

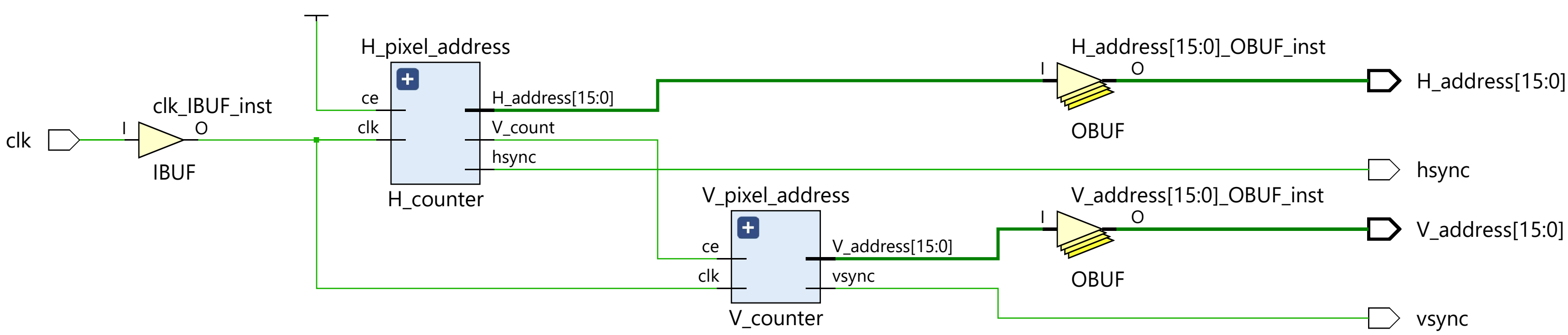
```
    //call insgtance of Colorize
    Colorize colors(

.ActiveRegion(ActiveRegion),
        .car(car),
        .road(road),
        .vgaRed(vgaRed),
        .vgaBlue(vgaBlue),
        .vgaGreen(vgaGreen)
    );


endmodule
```


play
clk
go
off_road
on_road
sec_2
ResetTimer
countdown
drive
flash_car
setup
drive_SM
0
0

```
`timescale 1ns / 1ps
/////////////////////////////////
/////////////////////////////////
////////////////
// Company:
// Engineer:
//
// Create Date: 03/05/2020
06:50:07 PM
// Design Name:
// Module Name: VGAcontroller
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////
/////////////////////////////////
////////////////

module VGAcontroller(
    input clk,
    output hsync,
    output vsync,
    output [15:0]H_address,
    output [15:0]V_address
    );

    //declare wire to hold
V_count output of H_counter
    wire V_count;
```

```
    //call instance of horizontal
and vertical address counters
    H_counter
H_pixel_address(.clk(clk),.ce(1'b
1),.hsync(hsync),.H_address(H_add
ress),.V_count(V_count));
    V_counter
V_pixel_address(.clk(clk),.ce(V_c
ount),.vsync(vsync),.V_address(V_
address));



endmodule
```

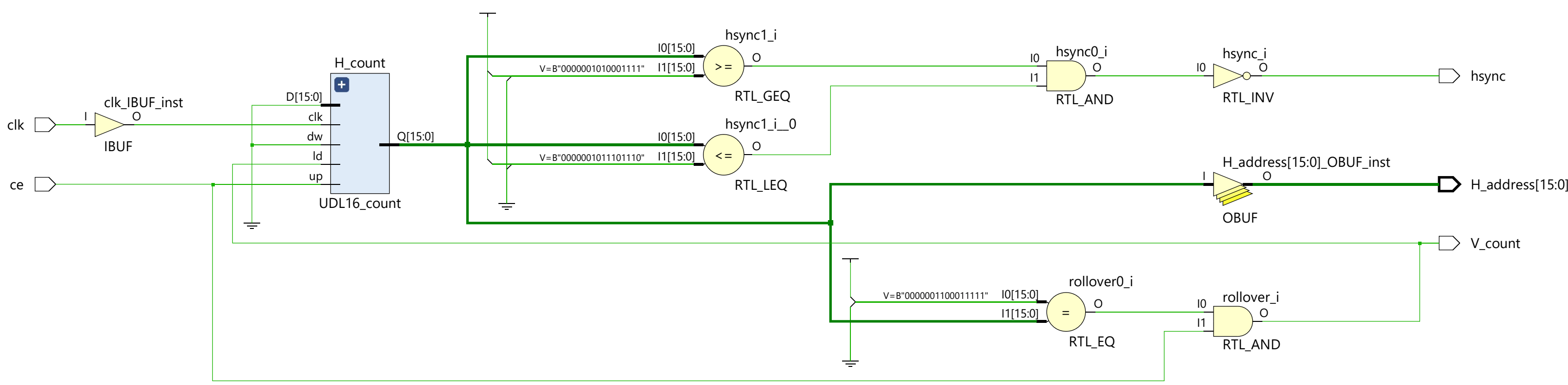

H_pixel_address
H_address[15:0]_OBUF_inst
clk_IBUF_inst
ce
H_address[15:0]
I
O
H_address[15:0]
clk
I
O
clk
V_count
OBUF
IBUF
hsync
hsync
H_counter
V_pixel_address
V_address[15:0]_OBUF_inst
I
O
V_address[15:0]
ce
V_address[15:0]
clk
vsync
OBUF
V_counter
vsync

```
`timescale 1ns / 1ps
////////////////////////////////
////////////////////////////////
////////////////
// Company:
// Engineer:
//
// Create Date: 03/05/2020
06:54:12 PM
// Design Name:
// Module Name: H_counter
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////
////////////////////////////////
////////////////

module H_counter(
    input clk,
    input ce,
    output hsync,
    output [15:0] H_address,
    output V_count
    );

    //declare wire to store
values for 16-bit counter
instance
```

```
    wire rollover, UTC, DTC;
    wire [15:0] h_MAX;

    //define horizontal max
address
    assign h_MAX = 16'd799;

    //call instance of 16-bit
counter
    UDL16_count
H_count(.D(16'd0),.up(ce),.dw(1'b
0),.ld(rollover),.clk(clk),.UTC(U
TC),.DTC(DTC),.Q(H_address));

    //define rollover signal
    assign rollover =
(h_MAX==H_address)&ce;

    //assign Hsync range
```

```
    assign hsync =
~((H_address>=16'd655)&(H_address
<=16'd750));

    //assign value to tell our
vertical address counter to count
    assign V_count = rollover;

endmodule
```



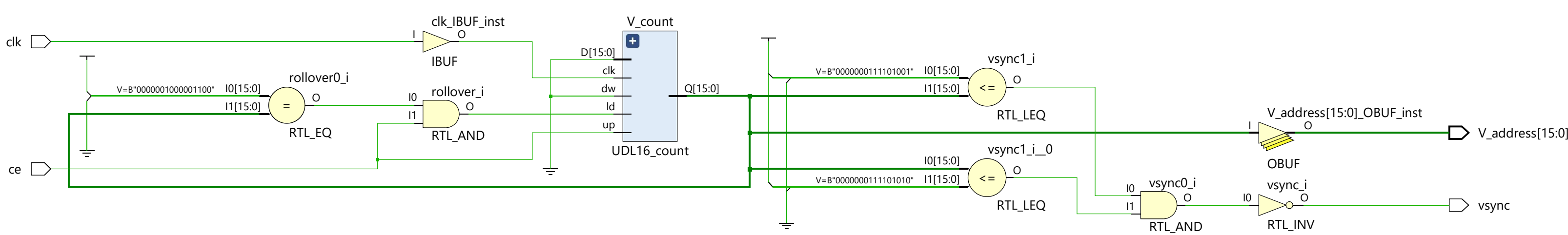
clk
ce
I
O
clk_IBUF_inst
IBUF
H_count
D[15:0]
clk
dw
ld
up
Q[15:0]
UDL16_count
hsync1_i
I0[15:0]
V=B"0000001010001111"
I1[15:0]
>=
O
RTL_GEQ
hsync1_i__0
I0[15:0]
V=B"0000001011101110"
I1[15:0]
<=
O
RTL_LEQ
hsync0_i
I0
I1
O
RTL_AND
hsync_i
I0
O
RTL_INV
hsync
H_address[15:0]_OBUF_inst
I
O
OBUF
H_address[15:0]
V_count
rollover0_i
V=B"0000001100011111"
I0[15:0]
I1[15:0]
=
O
RTL_EQ
rollover_i
I0
I1
O
RTL_AND

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////
//////////////////
// Company:
// Engineer:
//
// Create Date: 03/05/2020
06:54:12 PM
// Design Name:
// Module Name: V_counter
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////
////////////////////////////////
////////////////

module V_counter(
    input clk,
    input ce,
    output vsync,
    output [15:0] V_address
    );

    //declare wires for 16-bit
ocunter
    wire rollover, UTC, DTC;
    wire [15:0] v_MAX;
```

```
    //set wire to define max
vertical pixel address
    assign v_MAX = 16'd524;

    //call instance of 16-bit
counter
    UDL16_count
V_count(.D(16'd0),.up(ce),.dw(1'b
0),.ld(rollover),.clk(clk),.Q(V_a
ddress),.UTC(UTC),.DTC(DTC));

    //set wire to define when we
want to roll over back to 0
    assign rollover =
(v_MAX==V_address)&ce;

    //define VSync range
    assign vsync =
```

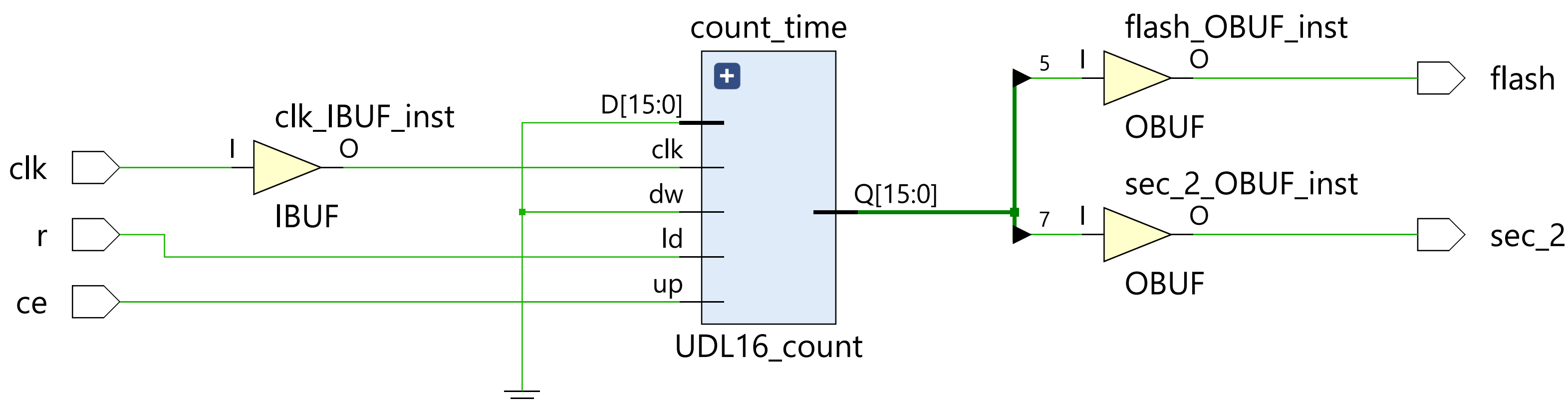
```
~((16'd489<=V_address)&(V_address
<=16'd490));

endmodule
```


clk
ce
clk_IBUF_inst
I
O
IBUF
V=B"0000001000001100"
I0[15:0]
I1[15:0]
rollover0_i
=
O
RTL_EQ
rollover_i
I0
I1
O
RTL_AND
V_count
D[15:0]
clk
dw
ld
up
Q[15:0]
UDL16_count
V=B"0000000111101001"
I0[15:0]
I1[15:0]
vsync1_i
<=
O
RTL_LEQ
vsync1_i__0
I0[15:0]
I1[15:0]
V=B"0000000111101010"
<=
O
RTL_LEQ
V_address[15:0]_OBUF_inst
I
O
OBUF
V_address[15:0]
vsync0_i
I0
I1
O
RTL_AND
vsync_i
I0
O
RTL_INV
vsync

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 03/07/2020
09:43:54 PM
// Design Name:
// Module Name: time_counter
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////
////////////////////////////////
////////////////


module time_counter(
    input clk,
    input ce,
    input r,
    output sec_2,
    output flash
    );

    //declare wires to store
16-bit counter outputs
    wire UTC, DTC;
```

```
    wire [15:0]time_bits;

    //call instance of 16-bit
counter
    UDL16_count
count_time(.clk(clk),.up(ce),.dw(
1'b0),.ld(r),.D({16{1'b0}}),.UTC(
UTC),.DTC(DTC),.Q(time_bits));

    //assign output values
    assign sec_2 = time_bits[7];
    assign flash = time_bits[5];

endmodule
```


count_time
D[15:0]
clk
dw
ld
up
UDL16_count
Q[15:0]
clk_IBUF_inst
I
O
IBUF
clk
r
ce
flash_OBUF_inst
5
I
O
OBUF
flash
sec_2_OBUF_inst
7
I
O
OBUF
sec_2

```
`timescale 1ns / 1ps
/////////////////////////////////
/////////////////////////////////
////////////////
// Company:
// Engineer:
//
// Create Date: 03/06/2020
12:06:57 AM
// Design Name:
// Module Name: Create_Obj
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////
//////////////////////////////////
////////////////


module Create_Obj(
    input [15:0] H_address,
    input [15:0] V_address,
    input clk,
    input frame,
    input L,
    input R,
    input drive,
    input setup,
    input flash,
    input flash_car,
```

```
    input [2:0]size,
    output ActiveRegion,
    output car,
    output off_road,
    output on_road,
    output [6:0]road
    );

    //define H_boundary and
V_boundary
    wire H_boundary, V_boundary;

    //assign boundaries of our
Active Region
    assign H_boundary =
H_address<=16'd639;
    assign V_boundary =
V_address<=16'd479;
```

```
    //assign ActiveRegion
    assign ActiveRegion =
H_boundary&&V_boundary;

    //define wires for x and y
bounds of our car
    wire car_x;
    wire car_y;

    //assign x and y bounds for
car
    assign car_x =
(H_address>=16'd312)&&(H_address<
=16'd327);
    assign car_y =
(V_address>=16'd393)&&(V_address<
=16'd408);

    //declare wire to assign car
```

```
flashing and car not flashing
values
    wire flashing_car;
    wire static_car;

    //assign static and
flashing_car values
    assign flashing_car =
car_x&car_y&flash;
    assign static_car =
car_x&car_y;

    //assign car object using a
2_1_mux to determine the output
    //as flashing or not
    m2_1 car_mux(

.in({flashing_car,static_car}),
        .sel(flash_car),
```

```
        .o(car)
    );

    //declare wire for road_1
    wire [15:0]road_h_address_0;
    wire [15:0]road_h_address_1;
    wire [15:0]road_h_address_2;
    wire [15:0]road_h_address_3;
    wire [15:0]road_h_address_4;
    wire [15:0]road_h_address_5;
    wire [15:0]road_h_address_6;

    //call an instance of our
road generator
    gen_road road_seg_0(
        .clk(clk),
        .drive(drive),
        .V_address(V_address),
        .H_address(H_address),
```

```
        .frame(frame),
        .INIT_H_POS(16'd280),

.PREV_H_POS(road_h_address_6),
        .INIT_V_POS(16'd559),
        .L(L),
        .R(R),
        .reset_road(setup),
        .size(size),
        .road(road[0]),

.road_h_address(road_h_address_0)
    );



    //call an instance of our
road generator
    gen_road road_seg_1(
        .clk(clk),
```

```
        .drive(drive),
        .V_address(V_address),
        .H_address(H_address),
        .frame(frame),
        .INIT_H_POS(16'd310),

.PREV_H_POS(road_h_address_0),
        .INIT_V_POS(16'd479),
        .L(L),
        .R(R),
        .reset_road(setup),
        .size(size),
        .road(road[1]),

.road_h_address(road_h_address_1)
    );



    //call an instance of our
```

```
road generator
    gen_road road_seg_2(
        .clk(clk),
        .drive(drive),
        .V_address(V_address),
        .H_address(H_address),
        .frame(frame),
        .INIT_H_POS(16'd326),

.PREV_H_POS(road_h_address_1),
        .INIT_V_POS(16'd399),
        .L(L),
        .R(R),
        .reset_road(setup),
        .size(size),
        .road(road[2]),

.road_h_address(road_h_address_2)
    );
```

```
    //call an instance of our
road generator
    gen_road road_seg_3(
        .clk(clk),
        .drive(drive),
        .V_address(V_address),
        .H_address(H_address),
        .frame(frame),
        .INIT_H_POS(16'd301),

.PREV_H_POS(road_h_address_2),
        .INIT_V_POS(16'd319),
        .L(L),
        .R(R),
        .reset_road(setup),
        .size(size),
        .road(road[3]),
```

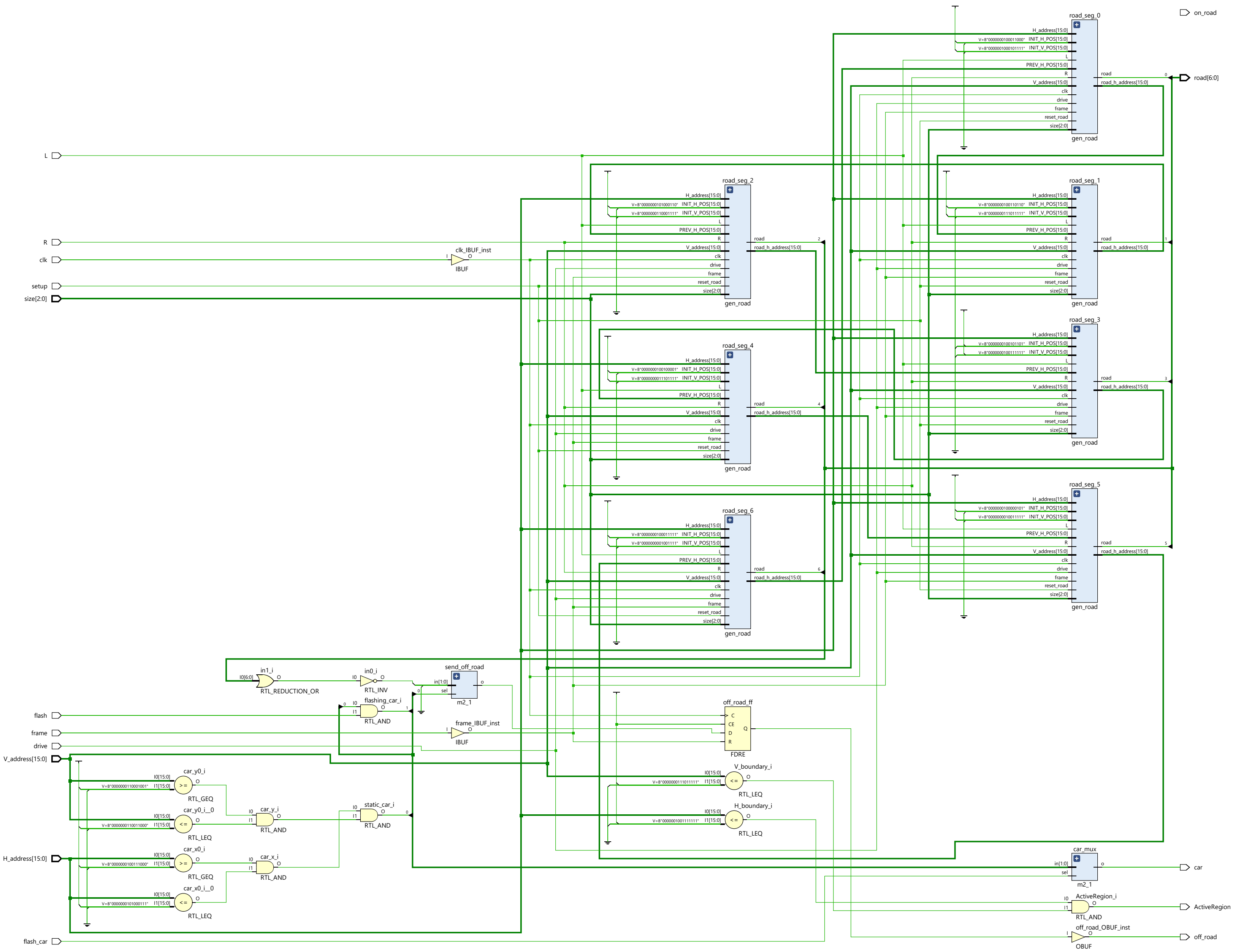
```
.road_h_address(road_h_address_3)
    );

    //call an instance of our
road generator
    gen_road road_seg_4(
        .clk(clk),
        .drive(drive),
        .V_address(V_address),
        .H_address(H_address),
        .frame(frame),
        .INIT_H_POS(16'd289),

.PREV_H_POS(road_h_address_3),
        .INIT_V_POS(16'd239),
        .L(L),
        .R(R),
        .reset_road(setup),
        .size(size),
```

```
        .road(road[4]),

.road_h_address(road_h_address_4)
    );

    //call an instance of our
road generator
    gen_road road_seg_5(
        .clk(clk),
        .drive(drive),
        .V_address(V_address),
        .H_address(H_address),
        .frame(frame),
        .INIT_H_POS(16'd261),

.PREV_H_POS(road_h_address_4),
        .INIT_V_POS(16'd159),
        .L(L),
        .R(R),
```

```
            .reset_road(setup),
            .size(size),
            .road(road[5]),

.road_h_address(road_h_address_5)
        );

        //call an instance of our
road generator
        gen_road road_seg_6(
            .clk(clk),
            .drive(drive),
            .V_address(V_address),
            .H_address(H_address),
            .frame(frame),
            .INIT_H_POS(16'd287),

.PREV_H_POS(road_h_address_5),
            .INIT_V_POS(16'd79),
```

```
        .L(L),
        .R(R),
        .reset_road(setup),
        .size(size),
        .road(road[6]),

.road_h_address(road_h_address_6)
    );

    wire off_road_w, on_road_w;

    //mux telling us when to
check for off_road signal
    m2_1
send_off_road(.in({~(|road),1'b0}
),.sel(static_car),.o(off_road_w)
);

    //call flip-flop to store
```

```
value of off_road once a frame
    FDRE #(.INIT(1'b0) )
off_road_ff (.C(clk), .R(frame),
.CE(1'b1), .D(off_road_w),
.Q(off_road));


endmodule
```



on_road
road[6:0]
L
R
clk
setup
size[2:0]
flash
frame
drive
V_address[15:0]
H_address[15:0]
flash_car
car
ActiveRegion
off_road
road_seg_0
road_seg_1
road_seg_2
road_seg_3
road_seg_4
road_seg_5
road_seg_6
gen_road
H_address[15:0]
INIT_H_POS[15:0]
INIT_V_POS[15:0]
PREV_H_POS[15:0]
V_address[15:0]
drive
reset_road
road
road_h_address[15:0]
clk_IBUF_inst
IBUF
frame_IBUF_inst
in1_i
RTL_REDUCTION_OR
in0_i
RTL_INV
send_off_road
m2_1
flashing_car_i
RTL_AND
off_road_ff
FDRE
V_boundary_i
H_boundary_i
RTL_LEQ
car_y0_i
RTL_GEQ
car_y0_i__0
car_y_i
static_car_i
car_x0_i
car_x_i
car_x0_i__0
car_mux
ActiveRegion_i
off_road_OBUF_inst
OBUF

```
`timescale 1ns / 1ps
////////////////////////////////
////////////////////////////////
////////////////
// Company:
// Engineer:
//
// Create Date: 03/06/2020
04:00:51 PM
// Design Name:
// Module Name: gen_road
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////
/////////////////////////////////
////////////////


module gen_road(
        input clk,
        input drive,
        input [15:0]V_address,
        input [15:0]H_address,
        input frame,
        input [15:0]INIT_H_POS,
        input [15:0]PREV_H_POS,
        input [15:0]INIT_V_POS,
        input L,
        input R,
```

```
        input reset_road,
        input [2:0]size,
        output road,
        output
[15:0]road_h_address
    );

    //declare wires to store
road_v_address for road segment 1
    wire [15:0]road_v_address;
    wire rollover;

    //call instance of
road_v_counter for road segment 1
    road_v_counter road_v_pos(
        .up(frame&drive),
        .reset(reset_road),
        .clk(clk),
        .INIT_V_POS(INIT_V_POS),
```

```
.road_v_address(road_v_address),
        .rollover(rollover)
     );

    //declare wires to hold road
position
    wire road_y, road_x;

    //assign road coordinate
ranges

    //declare wire to tell our
road to generate properly from
top of the screen to bottom
    wire sel;
    wire plus_80;
    wire minus_80;
```

```
    //define select
    assign sel =
(road_v_address<16'd80);

    //assign ranges for the
inputs
    assign plus_80 =
(V_address>=(road_v_address-16'd8
0))&&((road_v_address)>=V_address
);
    assign minus_80 =
V_address<=road_v_address;

    //use a mux to output the
correct coordinates
    m2_1 road_y_mux(
        .in({minus_80,plus_80}),
        .sel(sel),
        .o(road_y)
```

```
    );

    //declare wire to store rand
value
    wire [15:0]rnd;

    //call randomizer
    Random_LFSR
random_offset(.clk(clk),.ce(1'b1)
,.rnd(rnd[7:0]));

    //declare wire to store
inputs to Reset mux
    wire [15:0]ADD_OFFSET;
    wire [15:0]SUB_OFFSET;
    wire [15:0]RESET_H_POS;
    wire [15:0]OFFSET;

    //assign random offset and
```

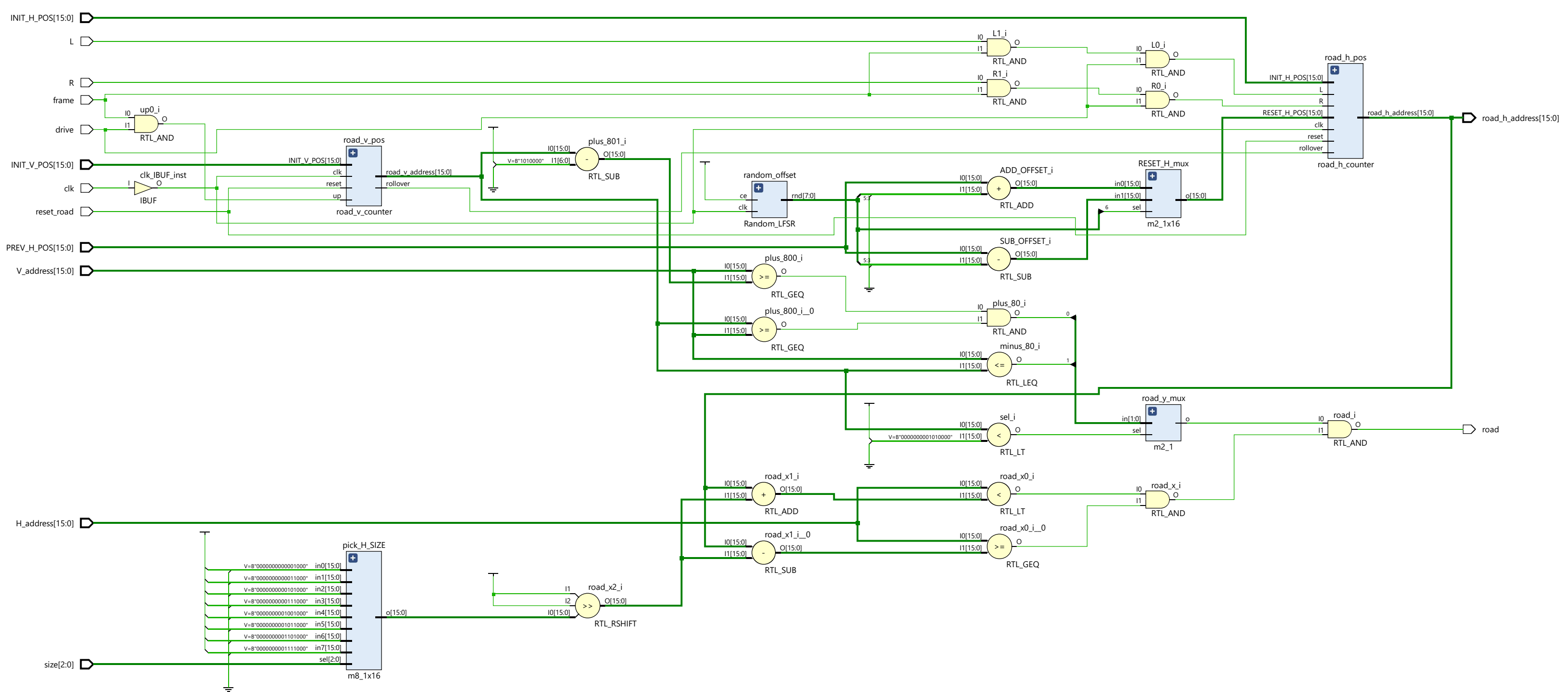
```
addition and subtraction of them
    assign OFFSET =
{10'd0,rnd[5:3],3'd0};
    assign ADD_OFFSET =
{PREV_H_POS+OFFSET};
    assign SUB_OFFSET =
{PREV_H_POS-OFFSET};

    //use a mux to determine the
proper reset position
    m2_1x16 RESET_H_mux(
        .in0(ADD_OFFSET),
        .in1(SUB_OFFSET),
        .sel(rnd[6]),
        .o(RESET_H_POS)
    );

    //call instance of
road_h_counter
```

```
    road_h_counter road_h_pos(
        .clk(clk),
        .L(L&frame&drive),
        .R(R&frame&drive),
        .reset(reset_road),
        .rollover(rollover),
        .INIT_H_POS(INIT_H_POS),

.RESET_H_POS(RESET_H_POS),

.road_h_address(road_h_address)
    );

    //declare wire to store
offset for H- address
    wire [15:0]H_OFFSET;
    wire [15:0]OFFSET_0;
    wire [15:0]OFFSET_1;
    wire [15:0]OFFSET_2;
```

```
    wire [15:0]OFFSET_3;
    wire [15:0]OFFSET_4;
    wire [15:0]OFFSET_5;
    wire [15:0]OFFSET_6;
    wire [15:0]OFFSET_7;


    //assign eight different
offset values
    assign OFFSET_0 = 16'd8;
    assign OFFSET_1 = 16'd24;
    assign OFFSET_2 = 16'd40;
    assign OFFSET_3 = 16'd56;
    assign OFFSET_4 = 16'd72;
    assign OFFSET_5 = 16'd88;
    assign OFFSET_6 = 16'd104;
    assign OFFSET_7 = 16'd120;

    //call instance of mux
```

```
    m8_1x16 pick_H_SIZE(
    .in0(OFFSET_0),
    .in1(OFFSET_1),
    .in2(OFFSET_2),
    .in3(OFFSET_3),
    .in4(OFFSET_4),
    .in5(OFFSET_5),
    .in6(OFFSET_6),
    .in7(OFFSET_7),
    .sel(size),
    .o(H_OFFSET)
    );

//    //assign static horizontal
boundaries for now
    assign road_x =
(H_address<(road_h_address+(H_OFF
SET>>1)))&&(H_address>=(road_h_ad
dress-(H_OFFSET>>1)));
```

```
    assign road = road_y&road_x;

//    assign onroad =


endmodule
```



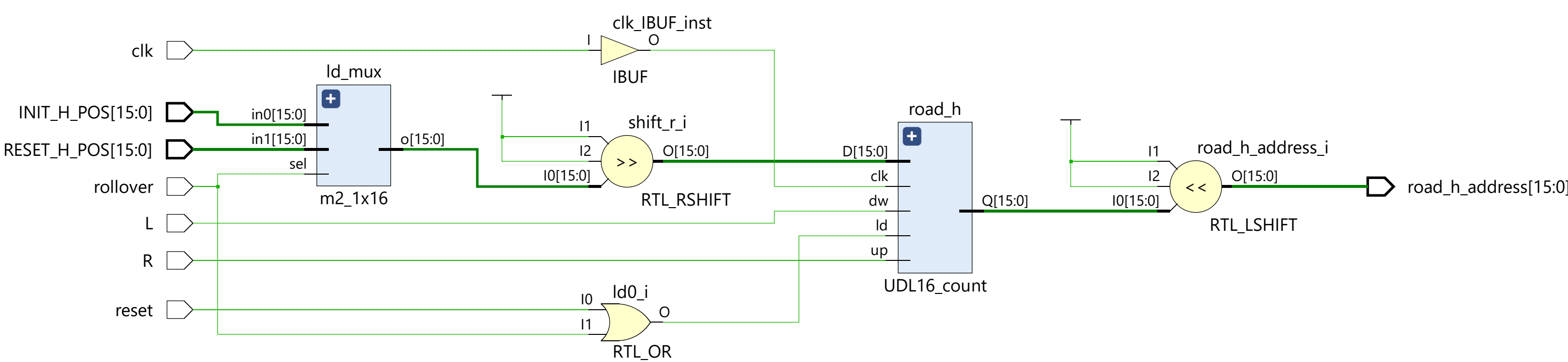
INIT_H_POS[15:0]
L
R
frame
drive
INIT_V_POS[15:0]
clk
reset_road
PREV_H_POS[15:0]
V_address[15:0]
H_address[15:0]
size[2:0]
up0_i
RTL_AND
clk_IBUF_inst
IBUF
road_v_pos
road_v_counter
road_v_address[15:0]
rollover
plus_801_i
RTL_SUB
random_offset
Random_LFSR
rnd[7:0]
L1_i
R1_i
L0_i
R0_i
ADD_OFFSET_i
RTL_ADD
SUB_OFFSET_i
RESET_H_mux
m2_1x16
road_h_pos
road_h_counter
road_h_address[15:0]
plus_800_i
RTL_GEQ
plus_800_i__0
plus_80_i
minus_80_i
RTL_LEQ
sel_i
RTL_LT
road_y_mux
m2_1
road_i
road
road_x1_i
road_x0_i
road_x_i
road_x1_i__0
road_x0_i__0
pick_H_SIZE
m8_1x16
road_x2_i
RTL_RSHIFT

```
`timescale 1ns / 1ps
////////////////////////////////
////////////////////////////////
////////////////
// Company:
// Engineer:
//
// Create Date: 03/06/2020
02:26:57 AM
// Design Name:
// Module Name: road_h_counter
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////
/////////////////////////////////
////////////////


module road_h_counter(
    input clk,
    input L,
    input R,
    input reset,
    input rollover,
    input [15:0]INIT_H_POS,
    input [15:0]RESET_H_POS,
    output [15:0]road_h_address
    );
```

```
    //declare wires to store
16-bit counter inputs/outputs
    wire UTC,DTC;
    wire [15:0]ld_bus;

    wire [15:0]shift_l;
    wire [15:0]shift_r;

    //call mux to deterine proper
load value
    m2_1x16 ld_mux(
        .in0(INIT_H_POS),
         .in1(RESET_H_POS),
         .sel(rollover),
         .o(ld_bus)
     );

    assign shift_r = ld_bus>>1;
```

```
    //call instance of 16-bit
counter
    UDL16_count road_h(
        .up(R),
        .dw(L),
        .ld(reset|rollover),
        .D(shift_r),
        .clk(clk),
        .UTC(UTC),
        .DTC(DTC),
        .Q(shift_l)
     );

    //assign output with shifted
bits to left
    assign road_h_address =
shift_l<<1;

endmodule
```


clk
INIT_H_POS[15:0]
RESET_H_POS[15:0]
rollover
L
R
reset
ld_mux
in0[15:0]
in1[15:0]
sel
o[15:0]
m2_1x16
clk_IBUF_inst
I
O
IBUF
shift_r_i
I1
I2
I0[15:0]
O[15:0]
RTL_RSHIFT
road_h
D[15:0]
clk
dw
ld
up
Q[15:0]
UDL16_count
ld0_i
I0
I1
O
RTL_OR
road_h_address_i
I1
I2
I0[15:0]
O[15:0]
RTL_LSHIFT
road_h_address[15:0]

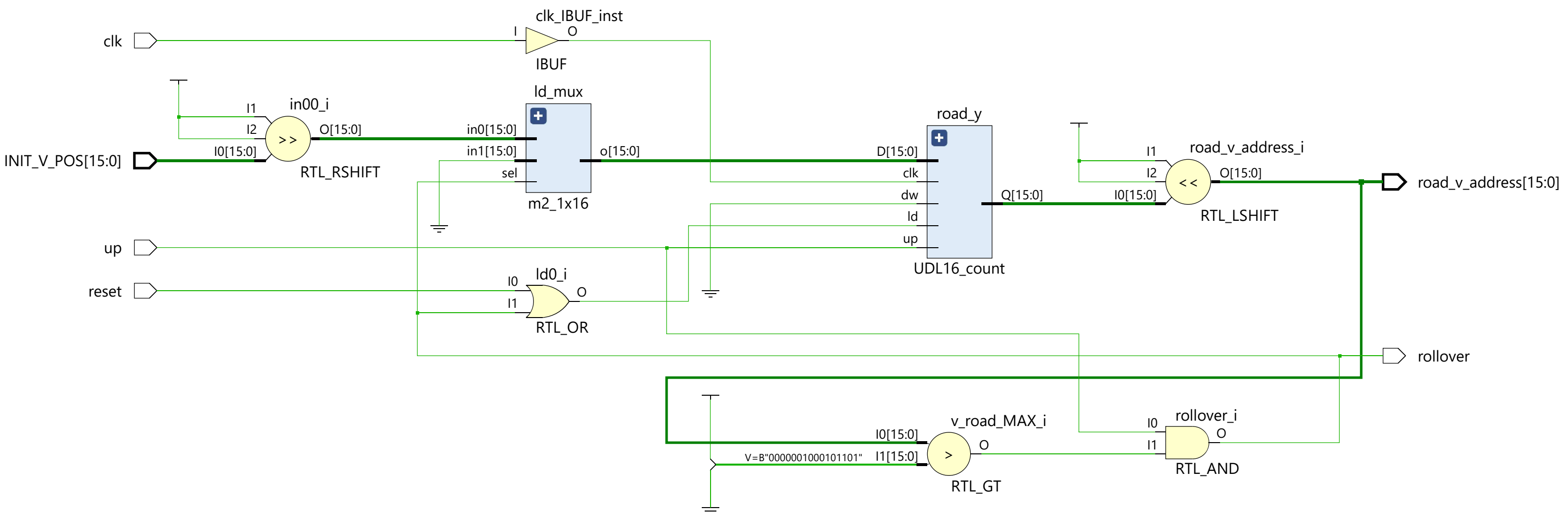
```
`timescale 1ns / 1ps
/////////////////////////////////
/////////////////////////////////
////////////////
// Company:
// Engineer:
//
// Create Date: 03/06/2020
02:26:57 AM
// Design Name:
// Module Name: road_v_counter
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////
////////////////////////////////
////////////////

module road_v_counter(
    input [15:0]INIT_V_POS,
    input up,
    input reset,
    input clk,
    output [15:0]road_v_address,
    output rollover
    );

    //declare wire to store
inputs/outputs of 16-bit counter
    wire UTC, DTC;
```

```
    wire [15:0]v_road_MAX;
    wire [15:0]ld_bus;

    //assign rollover value
    assign v_road_MAX =
road_v_address>16'd557;

    //assign rollover signal
    assign rollover =
up&(v_road_MAX);

    //call mux to determine load
value
    m2_1x16 ld_mux(
        .in0(INIT_V_POS>>1),
        .in1(16'd0),
        .sel(rollover),
        .o(ld_bus)
    );
```

```
    //wire to store unshifted
address
    wire [15:0]unshifted_address;

    //call instance of 16-bit
counter
    UDL16_count road_y(
        .up(up),
        .dw(1'b0),
        .ld(reset|rollover),
        .D(ld_bus),
        .clk(clk),
        .UTC(UTC),
        .DTC(DTC),
        .Q(unshifted_address)
    );

    //assign final output shifted
```

```
once to the left
    assign road_v_address =
unshifted_address<<1;

endmodule
```



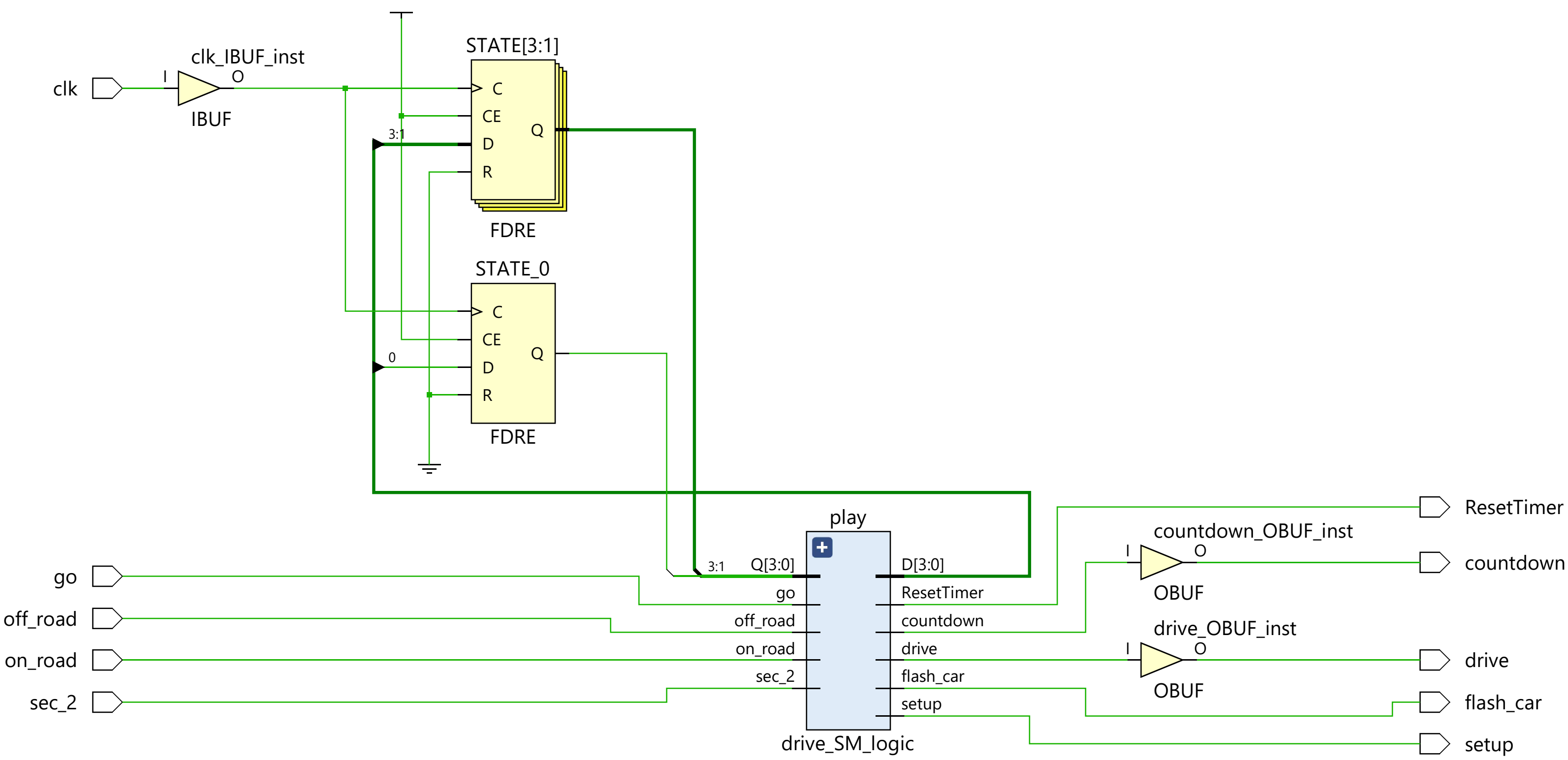
clk
clk_IBUF_inst
I
O
IBUF
in00_i
I1
I2
I0[15:0]
O[15:0]
RTL_RSHIFT
INIT_V_POS[15:0]
ld_mux
in0[15:0]
in1[15:0]
sel
o[15:0]
m2_1x16
road_y
D[15:0]
clk
dw
ld
up
Q[15:0]
UDL16_count
road_v_address_i
I1
I2
I0[15:0]
O[15:0]
RTL_LSHIFT
road_v_address[15:0]
up
reset
ld0_i
I0
I1
O
RTL_OR
rollover
v_road_MAX_i
I0[15:0]
I1[15:0]
V=B"0000001000101101"
O
RTL_GT
rollover_i
I0
I1
O
RTL_AND

```
`timescale 1ns / 1ps
/////////////////////////////////
/////////////////////////////////
////////////////
// Company:
// Engineer:
//
// Create Date: 03/06/2020
08:11:14 PM
// Design Name:
// Module Name: drive_SM
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////
////////////////////////////////
////////////////


module drive_SM(
    input clk,
    input go,
    input sec_2,
    input off_road,
    input on_road,
    output drive,
    output setup,
    output flash_car,
    output ResetTimer,
    output countdown
```

```
    //output [3:0]Q
    );

    //declare wire to store state
transitions
    wire [3:0]D;
    wire [3:0]Q;

    //call instance of
drive_logic
    drive_SM_logic play(
        .go(go),
        .sec_2(sec_2),
        .off_road(off_road),
        .on_road(on_road),
        .drive(drive),
        .setup(setup),
        .flash_car(flash_car),
        .ResetTimer(ResetTimer),
```

```
        .countdown(countdown),
        .D(D),
        .Q(Q)
    );

    FDRE #(.INIT(1'b1) ) STATE_0
(.C(clk), .R(1'b0), .CE(1'b1),
.D(D[0]), .Q(Q[0]));
    FDRE #(.INIT(1'b0) )
STATE[3:1] (.C({3{clk}}),
.R({3{1'b0}}), .CE({3{1'b1}}),
.D(D[3:1]), .Q(Q[3:1]));


endmodule
```

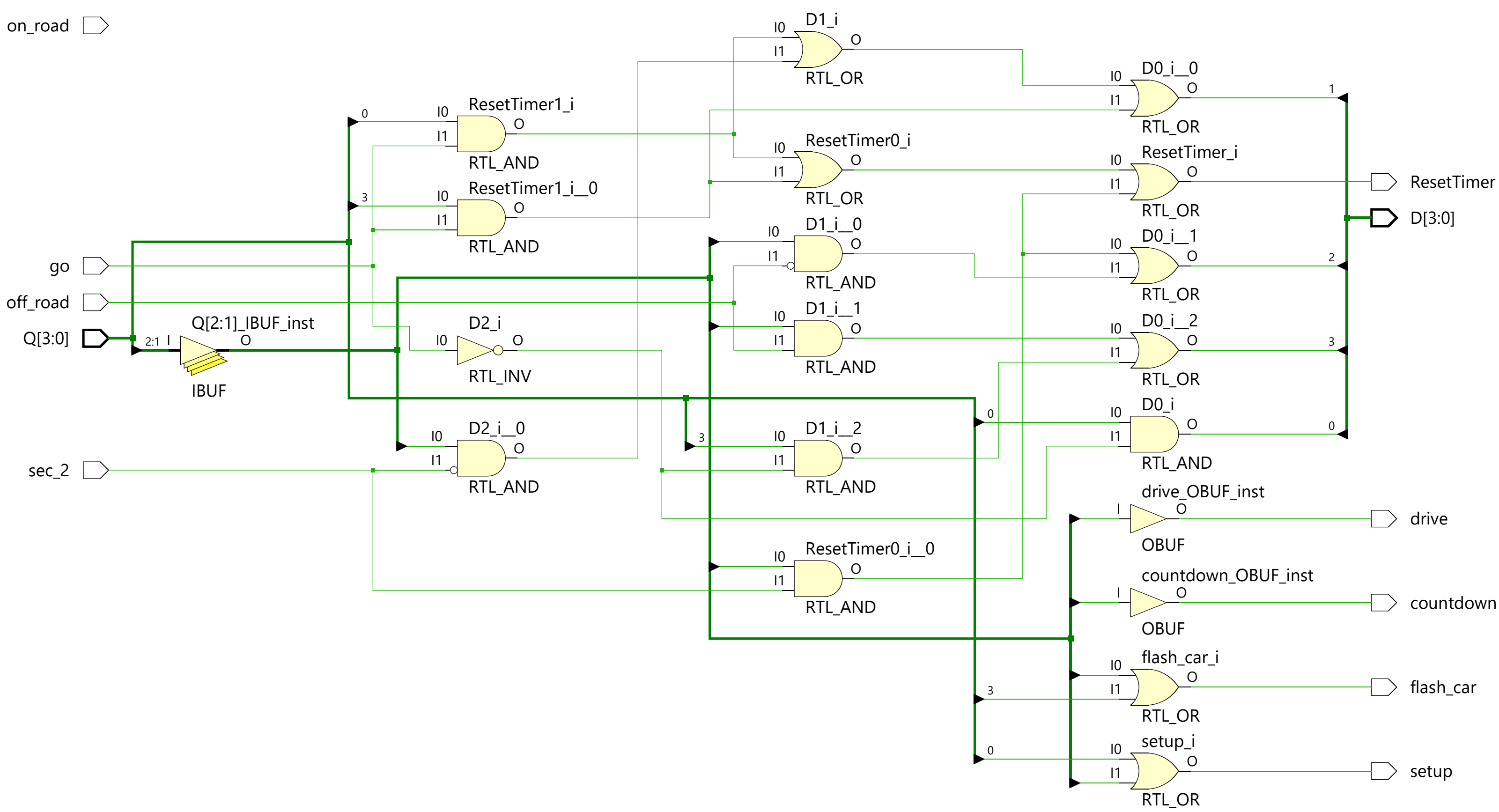

clk
clk_IBUF_inst
I
O
IBUF
STATE[3:1]
C
CE
D
R
Q
3:1
FDRE
STATE_0
C
CE
D
R
Q
0
FDRE
play
Q[3:0]
D[3:0]
3:1
go
off_road
on_road
sec_2
ResetTimer
countdown
drive
flash_car
setup
drive_SM_logic
countdown_OBUF_inst
OBUF
drive_OBUF_inst
OBUF
go
off_road
on_road
sec_2
ResetTimer
countdown
drive
flash_car
setup

```
`timescale 1ns / 1ps
////////////////////////////////
////////////////////////////////
////////////////
// Company:
// Engineer:
//
// Create Date: 03/06/2020
08:26:19 PM
// Design Name:
// Module Name: drive_SM_logic
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////
//////////////////////////////////
////////////////

module drive_SM_logic(
    input go,
    input sec_2,
    input on_road,
    input off_road,
    input [3:0]Q,
    output drive,
    output setup,
    output flash_car,
    output ResetTimer,
    output countdown,
```

```
    output [3:0]D
    );

    //assign next state logic
    assign D[0] = (Q[0]&(~go));
    //transition into D[1] from
D[2] b/c D[3] transition broken
    assign D[1] =
(Q[0]&go)|(Q[1]&(~sec_2))|(Q[3]&g
o);

    assign D[2] =
(Q[1]&sec_2)|(Q[2]&(~off_road));

    assign D[3] =
(Q[2]&off_road|Q[3]&(~go));
    //assign D[3] =
((Q[2]&(~on_road))|(Q[3]&(~go)));
```

```
    //assign outputs
    assign drive = Q[2];
    assign setup = Q[0]|Q[1];
    assign flash_car =
(Q[1]|Q[3]);
    assign ResetTimer =
(Q[0]&go)|(Q[3]&go)|(Q[1]&sec_2);
    assign countdown = Q[1];

endmodule
```

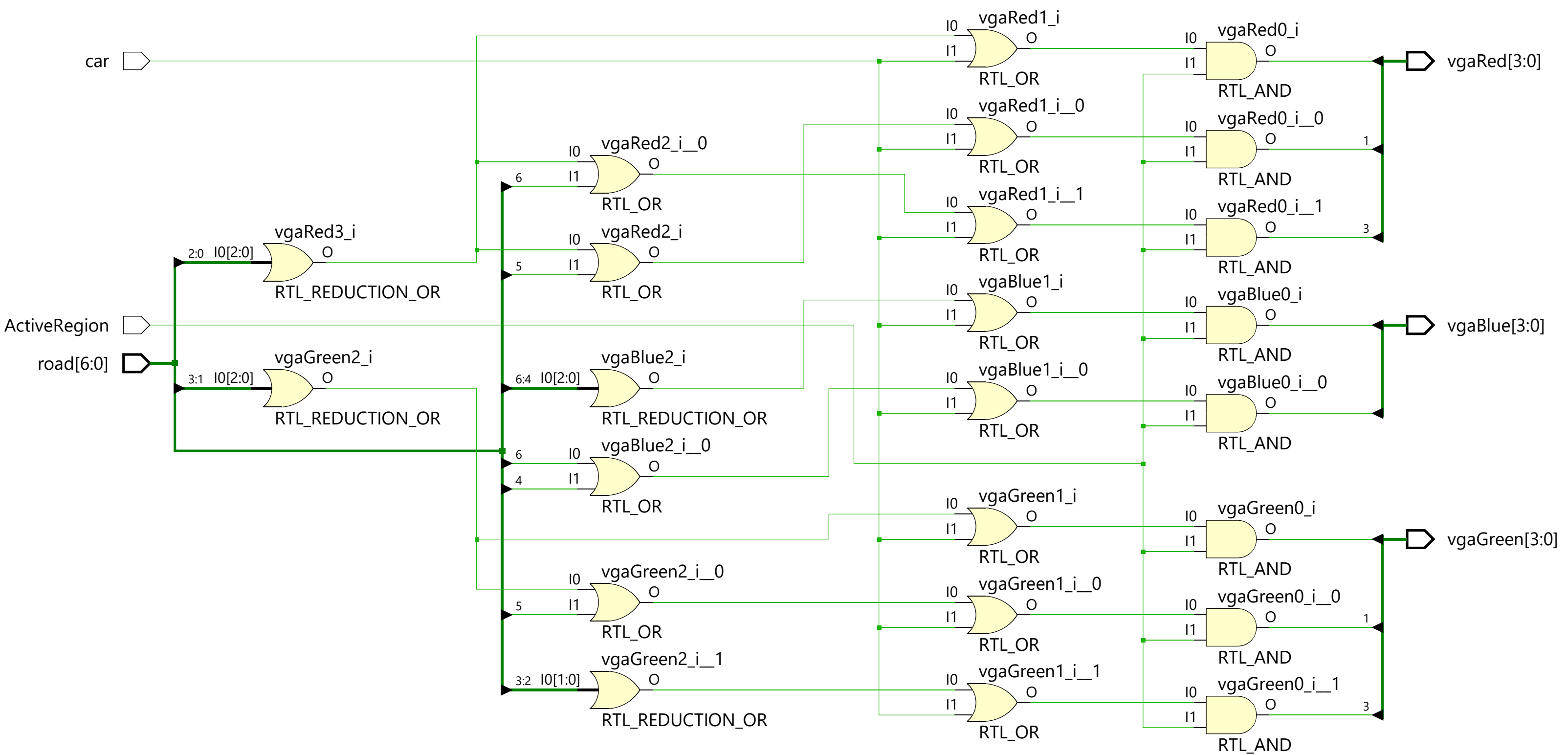

on_road
go
off_road
Q[3:0]
sec_2
Q[2:1]_IBUF_inst
IBUF
ResetTimer1_i
RTL_AND
ResetTimer1_i__0
RTL_AND
D2_i
RTL_INV
D2_i__0
RTL_AND
D1_i
RTL_OR
ResetTimer0_i
RTL_OR
D1_i__0
RTL_AND
D1_i__1
RTL_AND
D1_i__2
RTL_AND
ResetTimer0_i__0
RTL_AND
D0_i__0
RTL_OR
ResetTimer_i
RTL_OR
D0_i__1
RTL_OR
D0_i__2
RTL_OR
D0_i
RTL_AND
drive_OBUF_inst
OBUF
countdown_OBUF_inst
OBUF
flash_car_i
RTL_OR
setup_i
RTL_OR
ResetTimer
D[3:0]
drive
countdown
flash_car
setup

```
`timescale 1ns / 1ps
/////////////////////////////////
/////////////////////////////////
////////////////
// Company:
// Engineer:
//
// Create Date: 03/05/2020
10:57:14 PM
// Design Name:
// Module Name: Colorize
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////
/////////////////////////////////
////////////////


module Colorize(
    input ActiveRegion,
    input car,
    input [6:0]road,
    output [3:0]vgaRed,
    output [3:0]vgaBlue,
    output [3:0]vgaGreen
    );


    //new color assignations
```

```
    assign vgaRed[0] =
((|road[2:0])|car)&ActiveRegion;
    assign vgaRed[1] =
((|road[2:0]|road[5])|car)&Active
Region;
    assign vgaRed[2] =
((|road[2:0])|car)&ActiveRegion;
    assign vgaRed[3] =
((|road[2:0]|road[6])|car)&Active
Region;

    assign
vgaGreen[0]=((|road[3:1])|car)&Ac
tiveRegion;
    assign vgaGreen[1] =
((|road[3:1]|road[5])|car)&Active
Region;
    assign vgaGreen[2] =
((|road[3:1])|car)&ActiveRegion;
```

```
    assign vgaGreen[3] =
((|road[3:2])|car)&ActiveRegion;

    assign vgaBlue[0] =
((|road[6:4])|car)&ActiveRegion;
    assign vgaBlue[1] =
((road[6]|road[4])|car)&ActiveReg
ion;
    assign vgaBlue[2] =
((|road[6:4])|car)&ActiveRegion;
    assign vgaBlue[3] =
((road[6]|road[4])|car)&ActiveReg
ion;


endmodule
```


car
ActiveRegion
road[6:0]
vgaRed3_i
RTL_REDUCTION_OR
vgaGreen2_i
RTL_REDUCTION_OR
vgaRed2_i__0
RTL_OR
vgaRed2_i
RTL_OR
vgaBlue2_i
RTL_REDUCTION_OR
vgaBlue2_i__0
RTL_OR
vgaGreen2_i__0
RTL_OR
vgaGreen2_i__1
RTL_REDUCTION_OR
vgaRed1_i
RTL_OR
vgaRed1_i__0
RTL_OR
vgaRed1_i__1
RTL_OR
vgaBlue1_i
RTL_OR
vgaBlue1_i__0
RTL_OR
vgaGreen1_i
RTL_OR
vgaGreen1_i__0
RTL_OR
vgaGreen1_i__1
RTL_OR
vgaRed0_i
RTL_AND
vgaRed0_i__0
RTL_AND
vgaRed0_i__1
RTL_AND
vgaBlue0_i
RTL_AND
vgaBlue0_i__0
RTL_AND
vgaGreen0_i
RTL_AND
vgaGreen0_i__0
RTL_AND
vgaGreen0_i__1
RTL_AND
vgaRed[3:0]
vgaBlue[3:0]
vgaGreen[3:0]