ECE-540: SYSTEM ON CHIP DESIGN with FPGA

Road Rash-3D

Venkata Hemanth Tulimilli

Contents

OVERVIEW:

The game designed for this project was inspired by a street bike race arcade game that was popular in during the 1990's to the 2000 called Road Rash developed by Electronic Arts. The idea behind this multiplayer game is to get to the finish line by hook or by crook. Which meant players had the option to hit/kick the opponents to get them out of the race. It also provided to select from different maps for races on different terrains with an interesting story line.

Our project does not implement most of the fancy features available in the original game instead our design is a slightly modified version of the original game. In our version multiplayer options do not exist and instead of fighting with opponents we have randomly generated obstacles that must be dodged to reach the end line. If the player collides with the oncoming obstacle, then game over. The game maybe played from a span of 62 seconds to 10 minutes depending on the speed the bike traverses. Our version does not provide multiple maps instead we have given a 3D visualization as compensation. Also our version has a simple sound module designed for the bike and game termination only.

FINAL_MAIN:

This module is the main module of the game having many parallel always blocks running which give the position of the image. It takes clk, reset and push button inputs and passes the boke_col_position, car_icon_num, car_row, car_column and leds as outputs. The icon and the bcd module connect to this module for the input values. There are bike local parameters, car local parameters, clock divider parameters. Score icon and 6 score digits icon row and column start and end pixel values. The speed icon numbers for MPH row, column start end positions have been declared. Corresponding ROM's declarations and other wire declarations for local parameters have been made in this module.

RANDOM OBJECT GENERATOR:

The block generate 1Hz clock enable and 1/3 Hz clock enable and creating different obstacle combinations. On reset,tick_1_3hz,clk_cnt_1_3hz,count_init,obstacle_flag,left_obstacle_flag,right_obstacle_flag,middle_obstacle_flag are all set to zero. When clk_cnt_1hz == top_cnt_1hz, then tick1hz set high later tick_1_3hz set to high. Now 7 is added to 4 bit register to create different obstacle order. This covers an entire range of possible combinations of the 3 cars probability to appear. Individual bits of the adder are assigned to left, right and middle obstacle flags. A 5Hz, 10Hz, 20Hz, 40Hz, 50Hz frequency are generated to which correspond to different values of the bike speed.

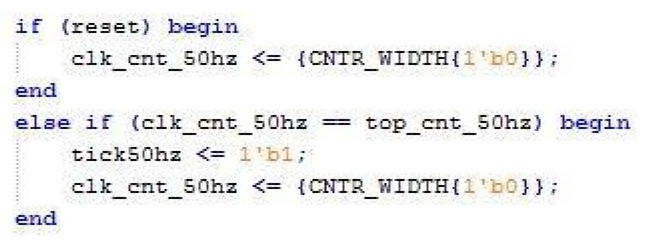
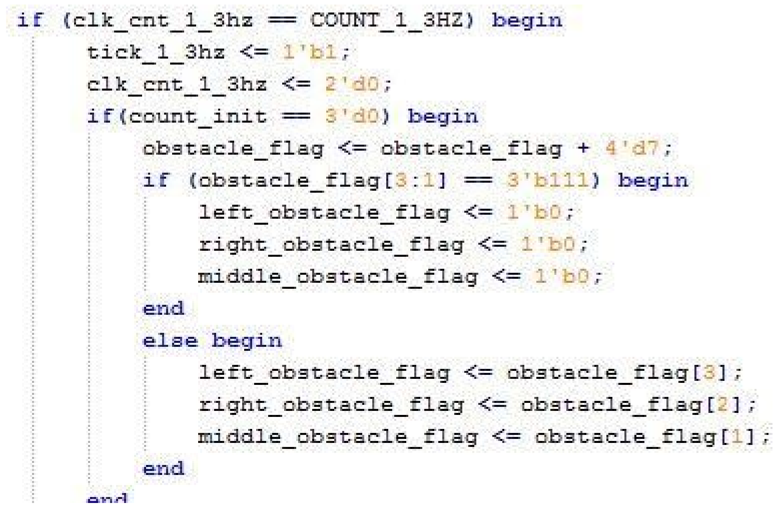

```
if (reset) begin
    clk_cnt_50hz <= {CNTR_WIDTH{1'b0}};
end
else if (clk_cnt_50hz == top_cnt_50hz) begin
    tick50hz <= 1'b1;
    clk_cnt_50hz <= {CNTR_WIDTH{1'b0}};
end
```

Fig: logic for random object generator module.

The acceleration block is used to calculate the speed based on the push button inputs. The inputs from the push buttons are read at 50Hz. Initially the acceleration flag nits are set to zero and upon 50Hz then inputs from debounce are tested if they are move forward, left , right or reverse and then acceleration flag is set. If it is in reverse mode, both acceleration flag and deceleration flag are set but priority is given to deceleration flag and the bike decelerates.



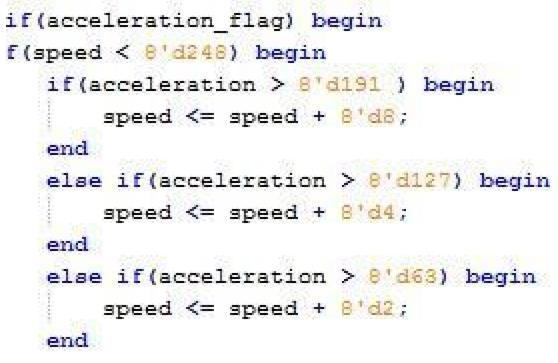
```
if (clk_cnt_1_3hz == COUNT_1_3HZ) begin
    tick_1_3hz <= 1'b1;
    clk_cnt_1_3hz <= 2'd0;
    if(count_init == 3'd0) begin
        obstacle_flag <= obstacle_flag + 4'd7;
        if (obstacle_flag[3:1] == 3'b111) begin
            left_obstacle_flag <= 1'b0;
            right_obstacle_flag <= 1'b0;
            middle_obstacle_flag <= 1'b0;
        end
        else begin
            left_obstacle_flag <= obstacle_flag[3];
            right_obstacle_flag <= obstacle_flag[2];
            middle_obstacle_flag <= obstacle_flag[1];
        end
    end
```

Fig: logic showing assignment of different bits as obstacle flags.

To make the game interesting, random obstacle generation was performed whose speed also increases along with the speed of the bike. A 4-bit adder was used to generate. From the 4 bits in the adder, each value is incremented by a value of 7. Now bit 2 corresponds to left car, bit 1 to middle car and bit 0 to right car. Later based on the value at the corresponding bit i.e. 1 for car generate and 0 for no car. This also gives the advantage to generate different combinations of cars to come in opposite directions. Adding 7 covers an entire range of values without missing any combination of 3 sequence generator.

SPEED CALCULATION BLOCK:

The speed of the bike is calculated in this block based on the range of acceleration register. If the deceleration is in between 0 and 63 then speed decremented by 8'd2. The same with acceleration, if >191 then speed is incremented by 8 or a different increment in other case based on its value at acceleration register.

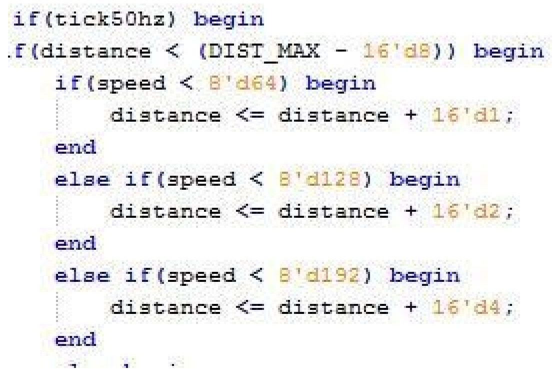


```
if(acceleration_flag) begin
f(speed < 8'd248) begin
   if(acceleration > 8'd191 ) begin
       speed <= speed + 8'd8;
   end
   else if(acceleration > 8'd127) begin
       speed <= speed + 8'd4;
   end
   else if(acceleration > 8'd63) begin
       speed <= speed + 8'd2;
   end
```

Fig: logic for speed calculation.

DISTANCE CALCULATION BLOCK MODULE:

The speed calculated from the speed block is passed at input to this and on 50Hz the signals are sampled. The equations for speed, acceleration and distance are derived from basic physics and stand correct throughout the race. according to the value of the DIST_MAX, the distance is incremented. left/middle/right car icon control signal always block. The display positions of the obstacles on the screen vary with he movement of the bike. The control number which takes care of the illusion effects is car_icon_num for displaying. The initial position of the car is set to display at the end of the road on the background screen image. Its position is incremented by row and column and also widening the size of the images as well This is responsible for the 3D effect for the obstacles approaching as we move further .



```
 if(tick50hz) begin
.f(distance < (DIST_MAX - 16'd8)) begin
    if(speed < 8'd64) begin
        distance <= distance + 16'd1;
    end
    else if(speed < 8'd128) begin
        distance <= distance + 16'd2;
    end
    else if(speed < 8'd192) begin
        distance <= distance + 16'd4;
    end
```

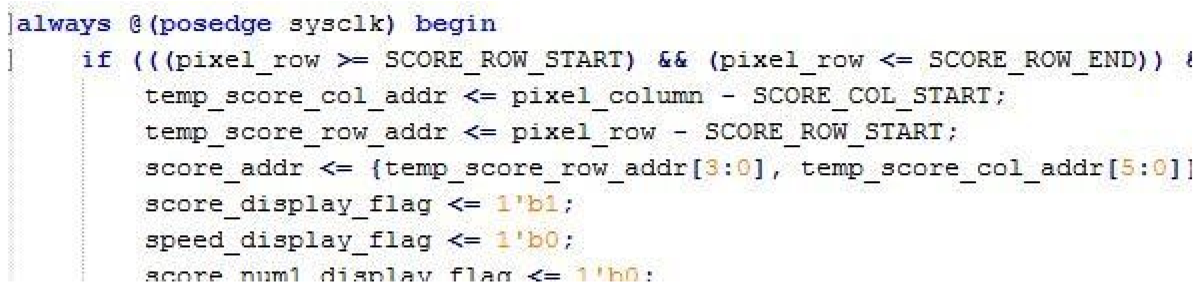
Fig: Logic showing distance calculation.

ICON MODULE:

The main screen display always block is responsible for the entire game image display as it sets and removes flags and is at the top of hierarchy. The module selects values that are to be displayed at each and every pixel based on flags set in other always blocks. The icons have separate always blocks where their flag bits are set. Upon bike_flag set the bike is displayed, this is always on except when a collision happens and game over image is displayed. The left_car_display_flag upon setting will select the left car to be displayed. Similar is the case with middle_car_display_flag and right_car_display_flag. The score_flag still remains even after the game ends to give the player information on his score and challenge other players. Separate flags for each bits number of the score are used.

```
if(score_display_flag == 1'b1) && (score_numxvalue != 8'd255)) begin //x is a number

    icon <= score_value; // the value is assigned.



else if collision flag is checked

else if car_display_flag

else if speed_numx_display_flag is checked.

else icon is assigned background_value.
```

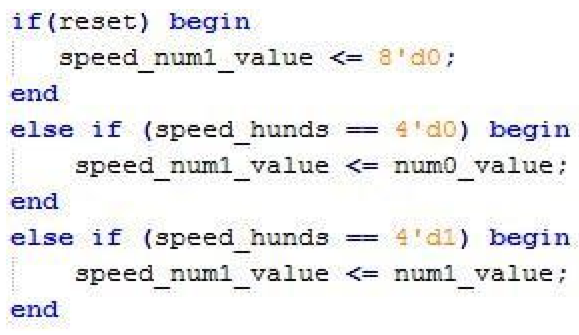
The always block for speed icon, score icon. mph icon and number address select calculates the address for the icons based on their pixel location on the screen according to 1024*768 display. The corresponding flags are set and cleared, the local parameters for this block can be found as local at the start of the final_main module. If the pixel row and column start, end values match with SCORE_ROW_START & END, then score_display_flag's are set. The score_addr is also updated. If the pixel_row, pixel_column is in between SCORE_NUM_ROW_START, END then temp_score_num_row and col_addr are given values. The pixel_row and pixel_column values are compared with SCORE_NUM_ROW_START and SCORE_NUM_ROW_END, there by assigning values to the temp_score_num_col_addr and temp_score_num_row_addr.



```
always @(posedge sysclk) begin
    if (((pixel_row >= SCORE_ROW_START) && (pixel_row <= SCORE_ROW_END)) &
        temp_score_col_addr <= pixel_column - SCORE_COL_START;
        temp_score_row_addr <= pixel_row - SCORE_ROW_START;
        score_addr <= {temp_score_row_addr[3:0], temp_score_col_addr[5:0]]
        score_display_flag <= 1'b1;
        speed_display_flag <= 1'b0;
        score_num1_display_flag <= 1'b0;
```

Fig: snippet of icon module setting flag bits.

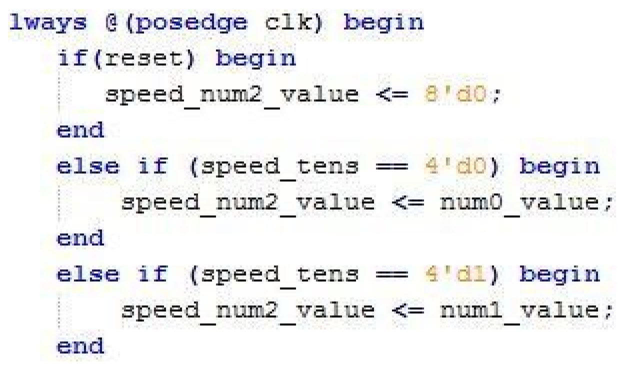
The speed, number, icon, value select block plays the role of control in choosing between what to display from the colour values. Selecting the colour values of the icon from corresponding ROM's based on speed_hunds value. This value is procured from the main block. The speed_hunds value is compared and corresponding num_value is assigned.



```
if(reset) begin
   speed_num1_value <= 8'd0;
end
else if (speed_hunds == 4'd0) begin
    speed_num1_value <= num0_value;
end
else if (speed_hunds == 4'd1) begin
    speed_num1_value <= num1_value;
end
```

Fig: speed_hunds comaprision and ROM assignment.

The speed, number2, icon value select block compares using the speed_tens values. Simultaneously the speed number3 icon value select block compares with speed_ones. The score number icon value selects which 8 bit colour value of number icon from different ROM's. The score blocks select the exact numerical values.



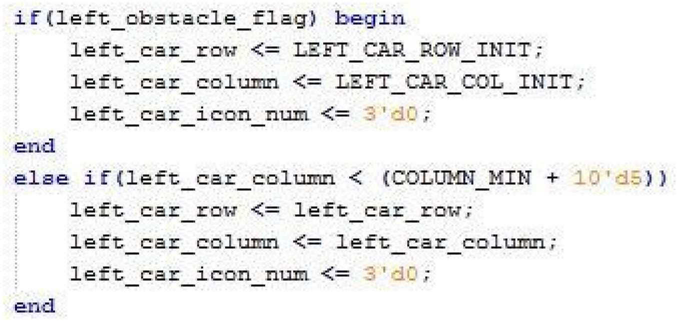
```
lways @(posedge clk) begin
   if(reset) begin
      speed_num2_value <= 8'd0;
   end
   else if (speed_tens == 4'd0) begin
       speed_num2_value <= num0_value;
   end
   else if (speed_tens == 4'd1) begin
       speed_num2_value <= num1_value;
   end
```

Fig: ROM selection

The collision detection block detects the collision by comparing pixel values of the bike_flag and car_display_flag. When both are set high and the pixel's match then there is collision happening, this is detected and game over image is displayed. If either one of the left, middle, right car flag positions match when their display values are also high then it is a collision. Background Icon value select block selects color value to be displayed in the background basing on whether a collision is happening or not.

```
if collision
        then display gameover_value
else    //depending on the background_num value
        background_value <=  backgroundx_value // where 'x' is one of the 3D images.
```

Left car icon value select block, selects the left car image value from the ROM's and passes it to left_car_value when the left_car_icon_num is equal to the corresponding decimal value. This now triggers left car icon display block to give address to left car ROM"S on left_icon_num and get the value at the pixel. Declaring different blocks for the left icon and select value, as well with right icon value and car icon display allows simultaneous running of the obstacle generation with respect to changes in the speed of bike from main module. Also this type of separate declaration help rendering a 3D illusion as compared to including all statements in a single block. The display values of the car icons keep increasing at every instant from 32*32 to 64*64 to 256*256 to give close to reality effect

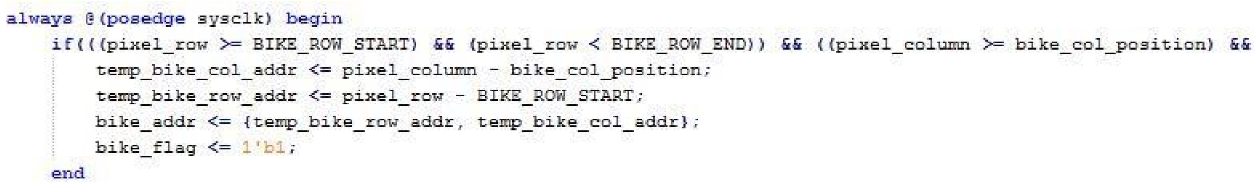


```
if(left_obstacle_flag) begin
    left_car_row <= LEFT_CAR_ROW_INIT;
    left_car_column <= LEFT_CAR_COL_INIT;
    left_car_icon_num <= 3'd0;
end
else if(left_car_column < (COLUMN_MIN + 10'd5))
    left_car_row <= left_car_row;
    left_car_column <= left_car_column;
    left_car_icon_num <= 3'd0;
end
```

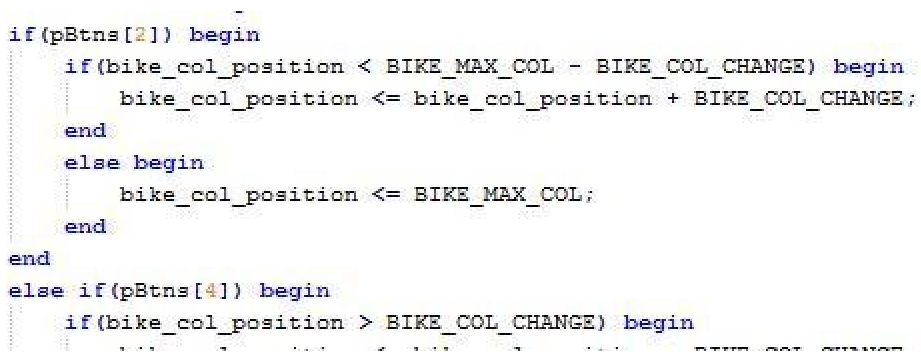
Fig: Car and position selection

BIKE ADDRESS SELECT BLOCK:

The pixels where the bike icon should be displayed is set and also the enable flag for bike display. The bike moves horizontally i.e. along the column of the screen so bike_col_postion is substracted from the pixel_column to determine the value of temp_bike_col_addr.



```
always @(posedge sysclk) begin
    if(((pixel_row >= BIKE_ROW_START) && (pixel_row < BIKE_ROW_END)) && ((pixel_column >= bike_col_position) &&
        temp_bike_col_addr <= pixel_column - bike_col_position;
        temp_bike_row_addr <= pixel_row - BIKE_ROW_START;
        bike_addr <= {temp_bike_row_addr, temp_bike_col_addr};
        bike_flag <= 1'b1;
    end
```



```
if(pBtns[2]) begin
    if(bike_col_position < BIKE_MAX_COL - BIKE_COL_CHANGE) begin
        bike_col_position <= bike_col_position + BIKE_COL_CHANGE;
    end
    else begin
        bike_col_position <= BIKE_MAX_COL;
    end
end
else if(pBtns[4]) begin
    if(bike_col_position > BIKE_COL_CHANGE) begin
```

Fig: Bike position logic snippet

ROM's DECLARED :

BACKGROUND1, BACKGROUND2, BACKGROUND3, BACKGROUND4, BACKGROUND5, BACKGROUND6, BACKGROUND7, BACKGROUND8, SCORE, SPEED, NUMO, NUM1, NUM2, NUM4, NUM5, NUM6, NUM7, NUM8, NUM9, MPH, GAMEOVER, BIKE, LCAR1, LCAR2, LCAR3, LCAR4, LCAR5, MIDCAR1, MIDCAR2, MIDCAR3, MIDCAR4, MIDCAR5, RIGHTCAR1, RIGHTCAR2, RIGHTCAR3, RIGHTCAR4, RIGHTCAR5

BCD_MODULE:

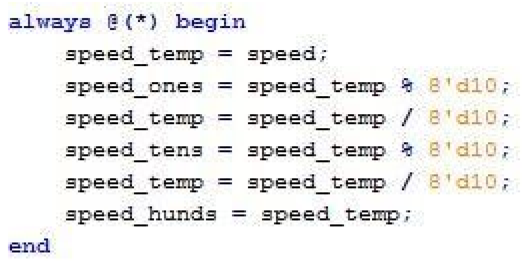
The need to display scores and speed led to the requirement of coding the decimal digits to binary format as a binary coded decimal. The speed is an 8 bit input to this module thereby covering $2^8 = 256$ decimal digits. Each digit is individually extracted as follows:

Ones place = speed % 10,

Tens place = (speed / 10) % 10,

Hundredths place = speed /100.

A similar approach was used to display the scores wherein the thoudandth, ten thousandth and even the hundred thousandth position are being displayed. The speed_temp and score_temp are the temporary register used for the calculations.

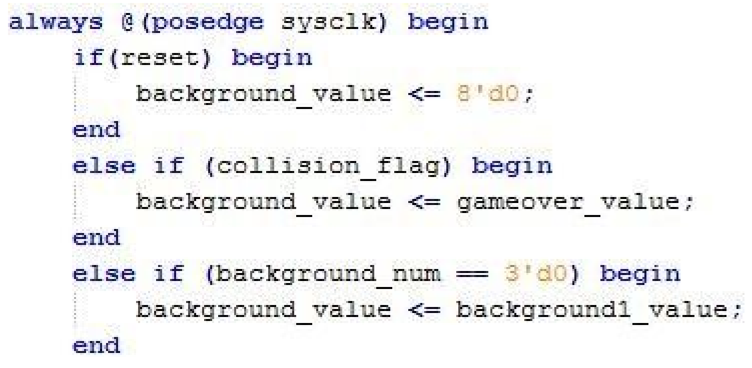


```
always @(*) begin
    speed_temp = speed;
    speed_ones = speed_temp % 8'd10;
    speed_temp = speed_temp / 8'd10;
    speed_tens = speed_temp % 8'd10;
    speed_temp = speed_temp / 8'd10;
    speed_hunds = speed_temp;
end
```

Fig: BCD implementation snippet

BACKGROUND:

Our main intention in designing the game was to implement the design as close as we could to the reality of bike moving on road. We have taken utmost care to design the images for the game making use of features from Adobe Photoshop and Illustrator. All the building were carefully sized to make every frame to illustrate bike maneuvering through real city having random cars as obstacles. Each frame had approaching buildings and incrementing sizes of buildings and background road. The close to reality effect was achieved by calling images based on the motion of the bike. Slower the speed, slower will be the increment of next images. Similar with with faster speeds. The frequency at which the images are called changes, thus giving the effect of 3D. Initially the images created were of the size 1024*768 but were later reduced in pixel to accommodate in the available memory space.



```
always @(posedge sysclk) begin
    if(reset) begin
        background_value <= 8'd0;
    end
    else if (collision_flag) begin
        background_value <= gameover_value;
    end
    else if (background_num == 3'd0) begin
        background_value <= background1_value;
    end
```

Fig: background or game over logic.

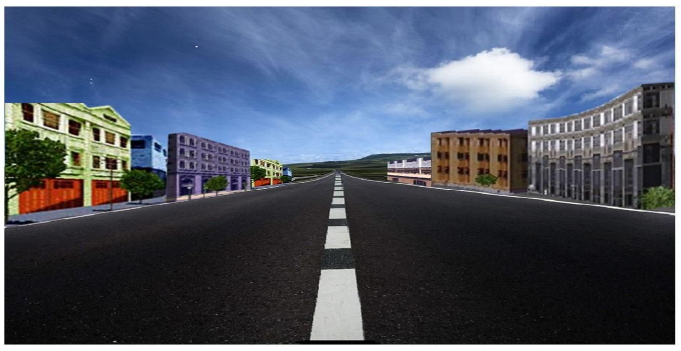


Fig: One of the 3D image backgrounds.

<u>PWM</u> for sound:

This module takes inputs as clk, reset, speed, duty_cycle from the top module and outputs pwm_audio_out. The top module gives the speed of the bike depending on the amount of time debounce switches are pressed. The speed frequency values ranges from 400Hz to 8050Hz from the equation.

speed_freq = (speed*OFFSET_MULT) + OFFSET_ADD;

This sets the counter and pwm values in this module. The main idea behind the variation is the frequency at which it is set when the bike is moving.

speed_freq = (speed*OFFSET_MULT) + OFFSET_ADD;

The top_count defines the frequency from

top_cnt = ((CLK_FREQUENCY_HZ / speed_freq) - 1);

On every posedge clk, either the counter is set to zero on reset or sets tickhz to specify the frequency of pwm signal or else it will increment clk_cnt and tickhz in the clk_divider block. The pwm generation block uses conditional statements to set the counter values and duty_cycle value. The tickhz values decide what values the pwm_audio_out has to take.

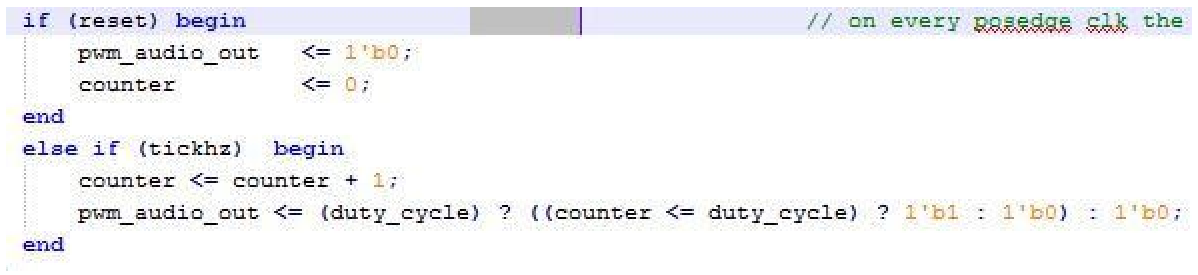

```
if (reset) begin                                      // on every posedge clk the
    pwm_audio_out   <= 1'b0;
    counter         <= 0;
end
else if (tickhz)  begin
    counter <= counter + 1;
    pwm_audio_out <= (duty_cycle) ? ((counter <= duty_cycle) ? 1'b1 : 1'b0) : 1'b0;
end
```

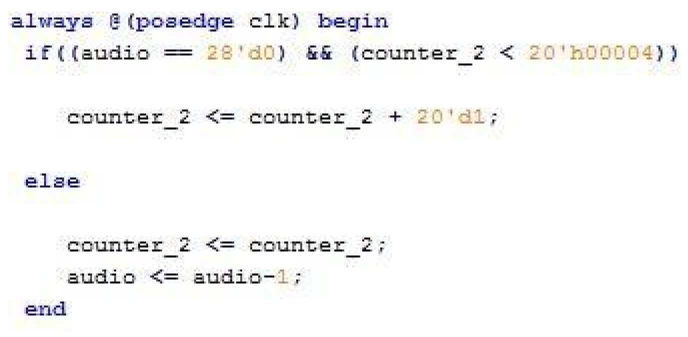
Fig: snippet showing pwm signal generation.

<u>GAME_OVER_PWM:</u>

This module for the game over sound. The audio bits 24 to 18 are used to generate sound.

pwm_temp = (audio[25] ? audio[24:18] : ~audio[24:18])

A ramp signal is created, which is set by the signal clock for ramp up or ramp down. The clk divider sets the values of the counter. Basically conuter_2 restricts the values of the counter_1. Audio signal values are obtained from clk divider. Infinite signal value is generated depending on counter values. We intend to restrict to 8 which binary values was achieved by restricting at less than 3'd4. If the counter_1 is set to 0, it gives 8 cycle, after which the generating of signals is stopped.



```
always @(posedge clk) begin
 if((audio == 28'd0) && (counter_2 < 20'h00004))

    counter_2 <= counter_2 + 20'd1;

 else

    counter_2 <= counter_2;
    audio <= audio-1;
 end
```

Fig: snippet showing counter implementation.

<u>DIFFICULTIES FACED:</u>

- ➢ Memory sizing: In the process of designing Road Rash similar to the actual game, we initially created 44 images to the size 1024*768 processing at nearly 3 frames per second giving a full 3D illusion. In addition there were images of bike, obstacles, score board mages to be allocated space. Due to limited size 512KB available on the fpga board, we have to consider two options in using the memory. Option 1 was reducing the size of the images and option 2 was creating a separate module for SD card and accessing images from the card.
- ➢ The reason why we could not use the SD card to hold images is the unavailability of a proper SD card host controller and though we did manage to get one from logic blocks we had licensing issues and when we tried to design the controller by ourselves there were portions of critical information that we could not get hold of such as the means to establish communication between the controller and the card over the available ports and so on.
- ➢ We scaled the images from 1024*768 to 256*192. This drastically pixilated the display but greatly reduced the size required for the images. Although we performed this, we also had to cut down to 8 images in total from the initial 44 images designed for the game. These 8 images were strategically designed to give the 3D effect and make the game a complete success.

REFERENCES:

1. http://www.fpga4fun.com/MusicBox.html

2.Xilinx manual for Nexys4 DDR.

3.ECE 540 Project 3 lecture slides by Roy Kravitz and Sarvesh Kulkarni.