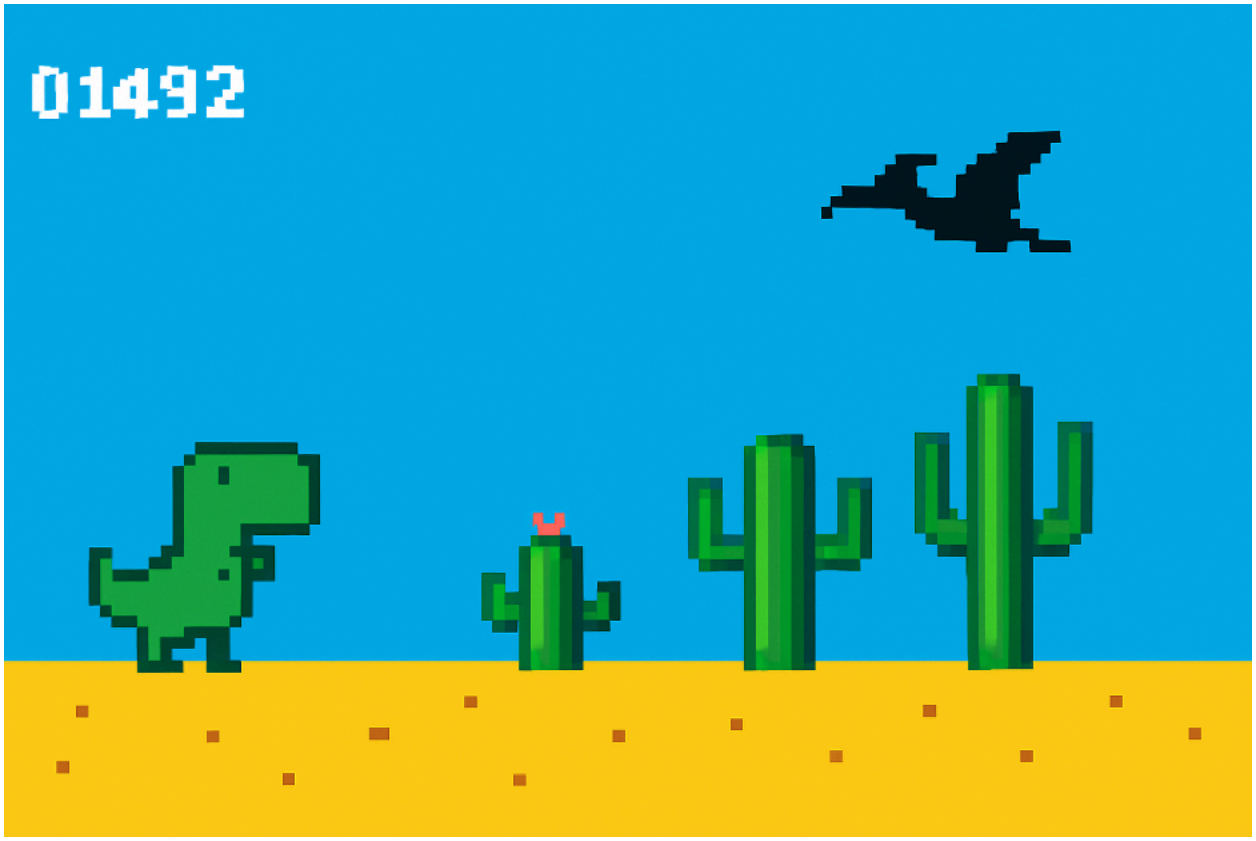
DINO RUN

Final Report for CSEE 4840 Embedded System Design

Swapnil Banerjee (sb5041), Roshan Prakash (rp3187), Anne Rose Sankar Raj (as7525)



Contents

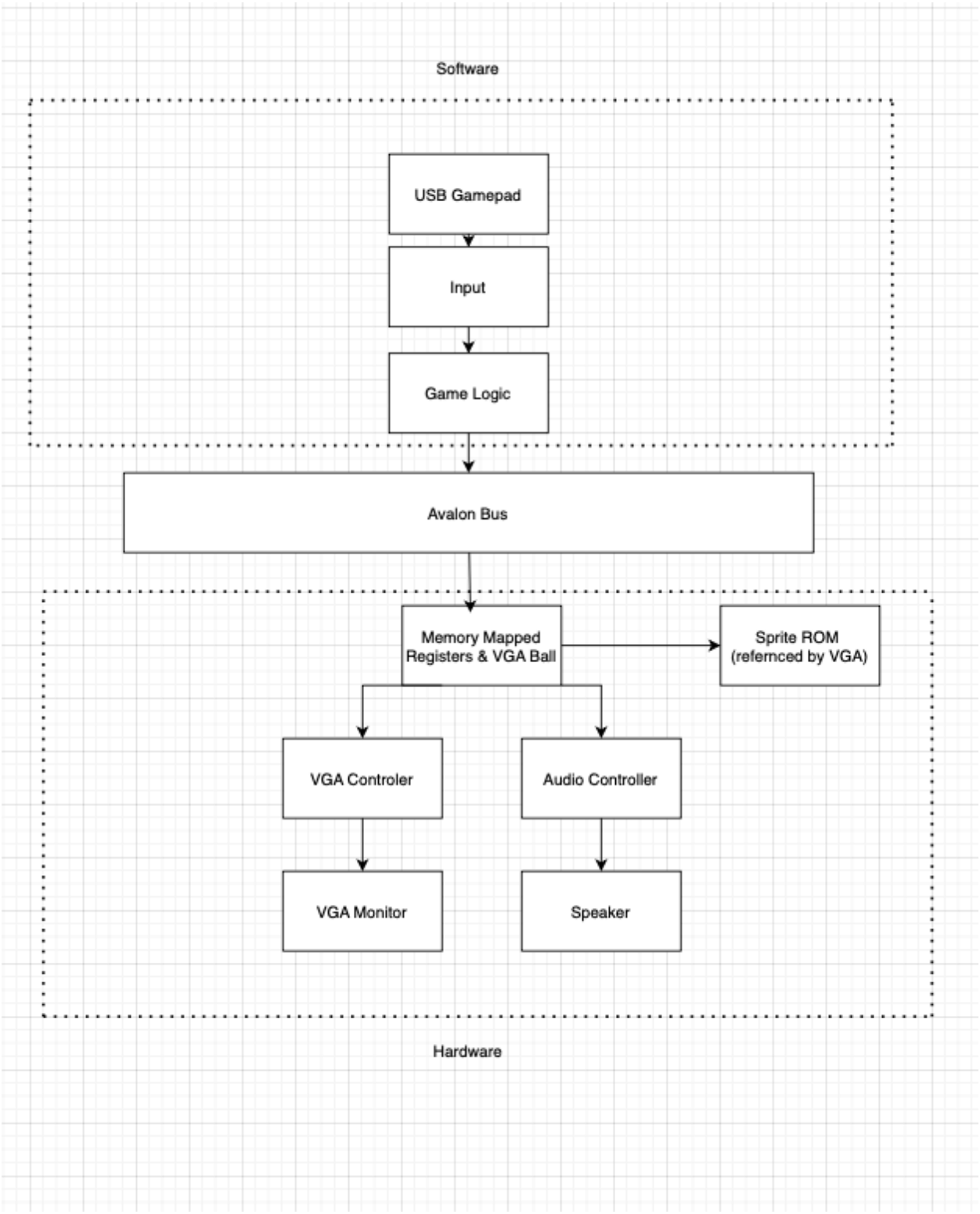
Overview:

Dino Run on the DE1-SoC is similar to the Google T-Rex runner but comes with our own tweaks. Our project uses an external Controller (Dragonrise Gamepad) for gaming and also has an audio background.

1. Hardware:.
    a. Sprites: Renders Sprites such as dinosaur, small cactus, cacti together, lava, pterodactyl, power-up, and replay using ROMs.
    b. Game Logic: Code handles obstacles, collisions, game speed up logics and has a “Godzilla mode” powerup. The game also switches between day and night backgrounds.
    c. Audio: Streams 16-bit samples to left/right channels.

2. Software:
    a. Memory-Mapped Input and Output: Maps registers for X,Y position of Dino and also duck, jump and replay mode.
    b. Controller: Reads USB gamepad inputs using libusb.
    c. Physics Loop: Integrates gravity and jumping, then writes updates into FPGA.

Basic Block Diagram:



Obstacle Movement Logic:

The obstacle positions are only updated every few milliseconds. Technically we are using a 50MHz clock so the obstacles should be moving 2,000,000/50,000,000 =0.04 seconds. Each obstacle moves in line to the left by a fixed number of pixels and resets to the right side of the screen (starting at 1280 pixels) when it hits the left edge. The new horizontal position of the obstacle is then calculated using a linear feedback shift register.

```
if (motion_timer >= 24'd2_000_000) begin
                                  s_cac_x <= (s_cac_x <=
obstacle_speed)
                             ? (HACTIVE + {lfsr,       4'd0})
                              : s_cac_x - obstacle_speed;
                 group_x <= (group_x <= obstacle_speed)
                              ? (HACTIVE + {lfsr ^ 6'h3F,4'd0})
                              : group_x - obstacle_speed;
                 lava_x  <= (lava_x  <= obstacle_speed)
                              ? (HACTIVE + {{lfsr[3:0]},6'd0})
                              : lava_x  - obstacle_speed;
                 ptr_x   <= (ptr_x   <= obstacle_speed)
                              ? (HACTIVE + {{lfsr[5:2]},6'd0})
                              : ptr_x   - obstacle_speed;
powerup_x <= (powerup_x <= obstacle_speed)
             ? (HACTIVE + {{lfsr[4:0]}, 5'd0})
             : powerup_x - obstacle_speed;
```

- Motion timer variable checks if 2,000,000 cycles has been reached (which happens every 0.04s).
- s_cac_x refers to the small cactus obstacle position (same for group_x => cacti together, lava_x => lava obstacle positon, ptr_x => pterodactyl position).
- If the obstacle_x position is less than the current obstacle speed that means it has crossed the left edge of the screen or it is at the left edge of the screen.
- If the above condition is true, then we push the obstacle to HACTIVE(1280) + offset. This offset is calculated using a Linear Shift Feedback register.
- If the condition is false, then change the obstacle_x position by obstacle_speed(set at 1 initially but keeps increasing with score).

Offset Calculation:

We use a linear feedback shift register to cycle through 63 numbers to give us a pseudo randomization. On each motion update the LFSR shifts its bits and reupdates using the XOR logic this makes sure that there is a new number that is generated every time. We then multiply the number by 16(for this sprite) which would mean that a new cactus spawns in 16 pixel increments somewhere between 1280 and 2288 pixels.

LFSR starts at a non-zero value (LFSR cannot start at 0 or else it will keep giving output as 0) and updates only if the game is not over.

Core of the LFSR logic –

```
lfsr <= { lfsr[4:0], lfsr[5] ^ lfsr[4] };
```

This represents the polynomial - (x^6) + (x^5) + 1
This 6-bit register cycles through all possible 6 bit values except 0 in a random order. Hence, fetching us a random offset.

Initial lfsr = 6'b101011
= b5 b4 b3 b2 b1 b0
= 1 0 1 0 1 1

feedback = b5 ^ b4 = 1 ^ 0 = 1

new_lfsr = { b4, b3, b2, b1, b0, feedback }
= { 0 , 1 , 0 , 1 , 1 , 1 }
= 6'b010111

Gameplay Logic:

The mechanics of the game involve ducking, jumping, power-ups and game reset. The user input is captured using a USB Dragonrise Gamepad. The software code will capture whether the user wants to jump, duck, or restart the game. These are actions are stored as boolean values and written to memory mapped registers–

```
#define DINO_Y_OFFSET      (1 * 4)
#define DUCKING_OFFSET     (13 * 4)
#define JUMPING_OFFSET     (14 * 4)
#define REPLAY_OFFSET      (19 * 4)
```

Jumping Physics:

Jumping is implemented in software code. When a jump is triggered, a negative initial velocity is applied and gravity is added to it each cycle. The resulting dino position is then written back into the dino_y_reg:

```
 *dino_y_reg = y_fixed >> FIXED_SHIFT;
```

The above line converts the fixed-point value back to an integer (screen pixel) before writing to hardware.

Obstacle Movement:

Obstacles move towards the left edge of the screen every 0.04 seconds which is controlled by a motion timer variable. When the obstacle hits the left edge of the screen, it resets to a position to the right of the screen (HACTIVE + calculated randomized offset).

Collision Detection:

The collision logic is handled in Verilog by bounding box comparisons between dino's position and each obstacle –

```
function automatic logic collide(
        input logic [10:0] ax, ay, bx, by,
        input logic [5:0]  aw, ah, bw, bh
    );
        return ((ax < bx + bw) && (ax + aw > bx) &&
                (ay < by + bh) && (ay + ah > by));
    endfunction
```

If the collision occurs and the dino is not in godzilla mode, the game enter the game-over state where the replay screen comes up —

```
 if (!godzilla_mode &&(collide(dino_x, dino_y, s_cac_x,  s_cac_y,
32,32,32,32) ||
                collide(dino_x, dino_y, group_x,  group_y,
64,32,32,32) ||
                collide(dino_x, dino_y, lava_x,   lava_y,
32,32,32,32) ||
                                collide(dino_x, dino_y, ptr_x,
ptr_y,    32,32,32,32))) begin
                game_over <= 1;
```

```
            end
```

The replay screen is displayed until the player presses the Start button.

In Godzilla mode, which is activated by collecting powerups, the dino becomes invincible and destroys obstacles on contact (pushes them back by 2000 pixels).

Background:

Day and night transition:

```
 // Night Timer Logic
            if (night_timer < 40'd1_500_000_000) begin
            night_timer <= night_timer + 1;  // Increment the timer
         end else if (night_timer == 40'd1_500_000_000) begin
            night_time <= ~night_time;
             night_timer <= 32'd0;
         end
            if (night_time) begin
                sky_r <= 8'd10;  // Dark blue night sky
                sky_g <= 8'd10;
                sky_b <= 8'd40;
            sun_r <= 8'd255; // White moon
            sun_g <= 8'd255;
            sun_b <= 8'd255;
            end else begin
                sky_r <= 8'd135;
                sky_g <= 8'd206;
                sky_b <= 8'd235; // Day sky color
                sun_r <= 8'd255; // Yellow sun
                sun_g <= 8'd255;
                sun_b <= 8'd0;
            end
```

We change between day and night by making use of a long timer.

Powerup Logic:

```
if (collide(dino_x, dino_y, powerup_x, powerup_y, 32, 32, 32, 32))
begin
    godzilla_mode <= 1;
    godzilla_timer <= 0;
    powerup_x <= 2000; // move off screen
end
            //Godzilla destroys
```

```
if (godzilla_mode) begin
    if (collide(dino_x, dino_y, s_cac_x, s_cac_y, 32, 32, 32, 32))
        s_cac_x <= 2000;
    if (collide(dino_x, dino_y, group_x, group_y, 64, 32, 32, 32))
        group_x <= 2000;
    if (collide(dino_x, dino_y, lava_x, lava_y, 32, 32, 32, 32))
        lava_x <= 2000;
    if (collide(dino_x, dino_y, ptr_x, ptr_y, 32, 32, 32, 32))
        ptr_x <= 2000;
end
```

Difficulty Progression:

```
if (s_cac_x <= obstacle_speed || group_x <= obstacle_speed ||
    lava_x  <= obstacle_speed || ptr_x   <= obstacle_speed) begin
    passed_count <= passed_count + 1;
end

if (passed_count >= 12) begin
    obstacle_speed <= obstacle_speed + 1;
    passed_count   <= 0;
end
```

Input Processing and Jump/Duck Physics:

```
bool want_jump = (y_axis == 0x00 && y == GROUND_Y);
bool want_duck = (y_axis == 0xFF && y == GROUND_Y);

if (want_jump) v = -12;
*jump_reg = want_jump;
*duck_reg = want_duck;

v += GRAVITY;
y += v;
if (y > GROUND_Y) { y = GROUND_Y; v = 0; }
*dino_y_reg = (uint32_t)y;
```

Audio:

Instantiated in our VGA ball module:

```
input  logic        L_READY,
input  logic        R_READY,
output logic [15:0] L_DATA,
output logic [15:0] R_DATA,
output logic        L_VALID,
output logic        R_VALID
```

Audio control–

```
if (sample_clock >= 285) begin  // 50MHz / 175550 ≈ 285
    sample_clock <= 0;
    audio_sample <= audio_data[audio_index];

    if (audio_index == 9659)
        audio_index <= 0;
    else
        audio_index <= audio_index + 1;
end else begin
    sample_clock <= sample_clock + 1;
end
```

Audio streaming–

```
if (L_READY) begin
    L_DATA  <= audio_sample;
    L_VALID <= (sample_clock == 0);
end else begin
    L_VALID <= 0;
end

if (R_READY) begin
    R_DATA  <= audio_sample;
    R_VALID <= (sample_clock == 0);
end else begin
    R_VALID <= 0;
end
```

Looping–

```
audio_index <= (audio_index == 9659) ? 0 : audio_index + 1;
```

Display Logic:

Each sprite is stored in a .rom module that works like a look-up table. These are synthesized are read-only memory blocks on the FPGA. Sprites like the Dino and obstacles are stored in these roms. To get each pixel in the sprite image, we use –

```
scac_sprite_addr <= (hcount - s_cac_x) + ((vcount - s_cac_y) * 32);
```

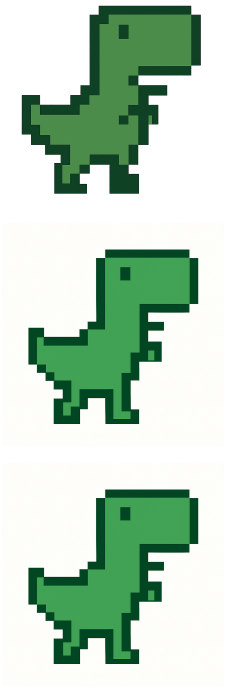
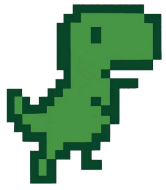
Sprite Drawing Logic:

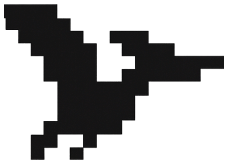
```
if (hcount >= dino_x && hcount < dino_x + 32 &&
    vcount >= dino_y && vcount < dino_y + 32) begin
    if (godzilla_mode)
        godzilla_sprite_addr <= (hcount - dino_x) + ((vcount -
dino_y) * 32);
    else
        dino_sprite_addr <= (hcount - dino_x) + ((vcount - dino_y) *
32);

    if (is_visible(dino_sprite_output)) begin
        a <= {dino_sprite_output[15:11], 3'b000};
        b <= {dino_sprite_output[10:5],  2'b00};
        c <= {dino_sprite_output[4:0],   3'b000};
    end
}
```

Background Logic:

```
if (vcount < 280) begin
    a <= sky_r;
    b <= sky_g;
    c <= sky_b;
} else if (vcount > 300) begin
    a <= 8'd100; // Darker brown ground stripe
    b <= 8'd40;
    c <= 8'd10;
} else begin
    a <= 8'd139; // Light brown ground
```

Sprites:

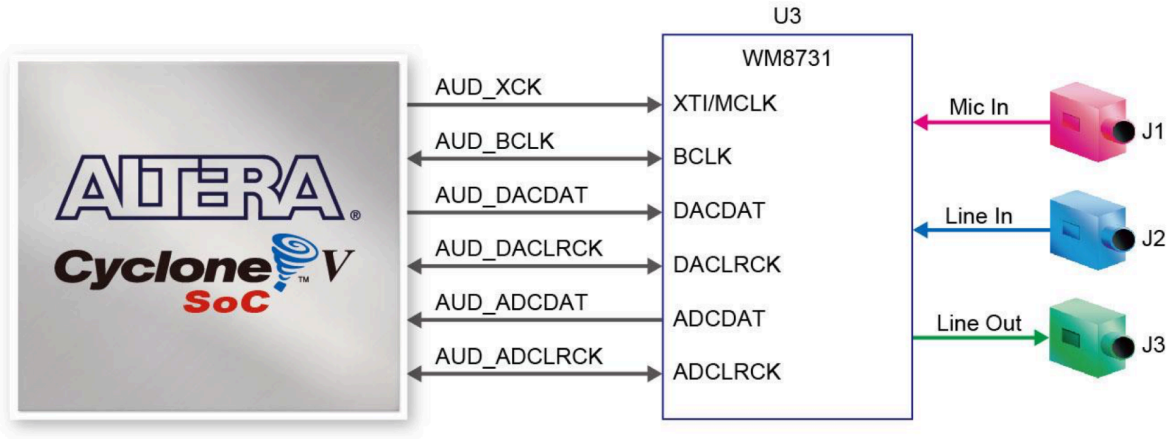
Category	Graphics	Size (bits)	Number of images	Total size (bits)
Dino		**32*32*16=16,384**	**3**	**49,152**
Dino Jump		**32*32*16=16384**	**1**	**16384**
Dino Duck		**32*32*16=16384**	**1**	**16384**

Godzilla		**32*32*16=16384**	**1**	**16384**
Small Cactus		**32*32*16=16384**	**1**	**16384**
Cacti Together		**64*32*16=32768**	**1**	**32768**
Lava		**32*32*16=16384**	**1**	**16384**
Powerup		**32*32*16=16384**	**1**	**16384**
Pterodactyl		**32*32*16=16384**	**2**	**32768**

Total				**212,592 bits**

Audio Logic:

The de1soc comes with a Woflson codec audio chip for handling 24-bit audio. This chip contains 2 ADCs and 2 DACs to interface with analog jacks and the FPGA with a digital interface. We will configure it via the I2C bus and provide a clock rate of 12MHz. We will only use of the DAC channels of the codec to transmit the signal for our Audio speaker. Altera provides two IP blocks for use to control the audio.



	Size	Samples	Total size (bits)
Background Music	**16**	**9660**	**16*9660 = 153600**
Total			**153600**

Controller Protocol:

Controller Mappings:

Nothing - 01 7f 7f 7f 7f 0f 00 00

Down - 01 7f 7f 7f ff 0f 00 00

Up - 01 7f 7f 7f 00 0f 00 00

Left - 01 7f 7f 00 7f 0f 00 00

Right - 01 7f 7f ff 7f 0f 00 00

Select - 01 7f 7f 7f 7f 0f 10 00

Start - 01 7f 7f 7f 7f 0f 20 00

D - 01 7f 7f 7f 7f 4f 00 00

A - 01 7f 7f 7f 7f 2f 00 00

Here is a visualization of how it works in our code. When we do report[4] we check the 4th row for 00 as that is what can be (assume that each 2 bit is a row). Then the c code recognized that we are giving the left operation since only the left operation has a 00 in the first row.

When a button is pressed the usb event is read in c via the libusb (usb based driver not kernel based). C program maps the input to the control flag ex jumping = 1>

Score:

Initializing Our Score:
Implemented using BCD (binary coded decimal), in this method we represent each decimal digit (0-9) usings its own binary value. For example 79 in BCD would be 0111 1001.

We start of by defining a BCD array:

```
logic [3:0] bcd [4:0];  // 5 digits: [0]=units, [1]=tens, ...,
[4]=ten-thousands
```

Where logic defined the width of each element and bcd defines the number of elements in the array. So our total storage would be 20 bits.

These values correspond directly with the decimal digits of the current score. For example, 54321 would be represented like so:

bcd[4] = 5
bcd[3] = 4
bcd[2] = 3
bcd[1] = 2
bcd[0] = 1

Incrementing Our Score:

Every time the player survives one tick (motion timer reaches 2 million cycles), we increment the score using a ripple carry mechanism.

```
bcd[0] <= bcd[0] + 1;
for (int i = 0; i < N_DIGITS-1; i++) begin
    if (bcd[i] == 4'd10) begin
        bcd[i]   <= 4'd0;
        bcd[i+1] <= bcd[i+1] + 1;
    end
end
if (bcd[N_DIGITS-1] == 4'd10)
    bcd[N_DIGITS-1] <= 4'd0;
```

We start off by incrementing the least significant bit bcd[0] which is the units place. If the incremented value equals 10 (which is not valid in BCD) we reset the digit to 0 and carry the 1 to the next digit which is the 10's place. We repeat this process up the array similar to how decimal addition works in real life. This process is extremely efficient compared to a modulo using operation like we previously had.

Score Display:

In order to display our score we use a font rom [10][8]

10 digits with 8 rows per digit

Essentially an array with each row representing 8 horizontal pixels. Each pixel is controlled by 1 bit meaning that 1 draws a black pixel and 0 defaults to the background color. When the raster scan reaches the score region of the screen we compute which digit is being drawn by dividing the horizontal pixel offset by 8. We then ensure that the most significant digit is drawn to the left by using bcd[N_DIGITS - 1 - idx] and the [7 - cx] term in order to flip the horizontal direction.

SCORE_X and SCORE_Y are the start positions

Code block:

```
rx  = hcount - SCORE_X;
idx = rx / 8;       // Which digit we're rendering (0-4)
cx  = rx % 8;       // Which column in the digit
ry  = vcount - SCORE_Y;  // Which row in the digit
```

Example:

SCORE_X = 120, SCORE_Y = 10

hcount = 144, vcount = 12

Then:

- rx = 144 - 120 = 24
- idx = 24 / 8 = 3
- cx = 24 % 8 = 0
- ry = 12 - 10 = 2

Example Pixel bitmap for digit 3:

font_rom[3] =

```
  8'b00111100,
  8'b01100110,
  8'b00000110,
  8'b00011100,
  8'b00000110,
  8'b01100110,
  8'b00111100,
  8'b00000000
```

Resetting the Array:

Finally our score is reset when the replay button is pressed using this logic in which the number in each place is set to 0:

```
for (int i = 0; i < N_DIGITS; i++) begin
    bcd[i] <= 4'd0;end
```

Register Address Mapping:

Address	Name	Size	Description
0	dino_x	10 bits	X position of the dino
1	dino_y	10 bits	Y position of the dino
13	ducking	1 bit	Set duck mode
14	jumping	1 bit	Set Jump mode
19	replay_button	1 bit	To trigger replay state

Gamepad Controller:

Conclusion:

We were able to create the Dino run game that we planned using the De1-soc. The jumping physics combined well with the positions of the sprites. We combined sprites, day-night transition, collision logic, audio playback and user input based jumping, ducking and restart of the game. Some challenges we faced were actually the fmax clock be clocked at 38 MHz (which is below ideal 50 MHz) which made some of the sprites to shimmer slgihtly. To reduce this we can reduce the load on the hardware side by implementing some features like day-night transition on the software side.

Complete codes:

vga_ball.sv:

```
// Dino Run code
module vga_ball(
    input  logic        clk,
    input  logic        reset,
input logic [31:0]  writedata,
    input logic         write,
    input               chipselect,
    input logic [8:0]   address,
    input  logic [7:0]  controller_report,

    output logic [7:0]  VGA_R, VGA_G, VGA_B,
    output logic        VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
VGA_SYNC_n,



    input  logic        L_READY,
input  logic        R_READY,
output logic [15:0] L_DATA,
output logic [15:0] R_DATA,
output logic        L_VALID,
output logic        R_VALID

);


    localparam HACTIVE = 11'd1280;
    localparam SCORE_X = 120;
    localparam SCORE_Y = 10;
     logic replay_button;
```

```
    initial begin
    $readmemh("background.hex", audio_data);
end



//audio sample variables

logic [15:0] audio_data[0:9659];
logic [17:0] audio_index;
logic [15:0] sample_clock;
logic [15:0] audio_sample;



    // VGA TIMING
    logic [10:0] hcount;
    logic [9:0]  vcount;

      //Score as 5-digit BCD
   localparam int N_DIGITS = 5;
   logic [3:0] bcd [N_DIGITS-1:0];  // [0]=units … [4]=ten-thousands

  logic [16:0]  score;
  logic [3:0]  digit_ten_thou, digit_thou, digit_h, digit_t, digit_u;
    logic [7:0]  font_rom [0:9][0:7];
     // Frame Animation
    logic [23:0] frame_counter;
    logic [1:0]  sprite_state;

    logic [10:0] cloud_offset;
    logic [23:0] cloud_counter;

    logic [7:0] sky_r, sky_g, sky_b;
    logic [23:0] sky_counter;
    logic [3:0]  sky_phase;
     logic [9:0] rx;
    logic [2:0] idx, ry;
    logic [3:0] cx;

 // Sun color
    logic [7:0] sun_r, sun_g, sun_b; //  sun color variables
```

```
    initial begin
        // simple 8×8 font for digits 0-9
        font_rom[0] =
'{8'h3C,8'h66,8'h6E,8'h7E,8'h76,8'h66,8'h3C,8'h00};
        font_rom[1] =
'{8'h18,8'h38,8'h18,8'h18,8'h18,8'h18,8'h7E,8'h00};
        font_rom[2] =
'{8'h3C,8'h66,8'h06,8'h1C,8'h30,8'h66,8'h7E,8'h00};
        font_rom[3] =
'{8'h3C,8'h66,8'h06,8'h1C,8'h06,8'h66,8'h3C,8'h00};
        font_rom[4] =
'{8'h0C,8'h1C,8'h2C,8'h4C,8'h7E,8'h0C,8'h0C,8'h00};
        font_rom[5] =
'{8'h7E,8'h60,8'h7C,8'h06,8'h06,8'h66,8'h3C,8'h00};
        font_rom[6] =
'{8'h3C,8'h66,8'h60,8'h7C,8'h66,8'h66,8'h3C,8'h00};
        font_rom[7] =
'{8'h7E,8'h06,8'h0C,8'h18,8'h30,8'h30,8'h30,8'h00};
        font_rom[8] =
'{8'h3C,8'h66,8'h66,8'h3C,8'h66,8'h66,8'h3C,8'h00};
        font_rom[9] =
'{8'h3C,8'h66,8'h66,8'h3E,8'h06,8'h66,8'h3C,8'h00};
    end
    localparam [7:0] FG_R = 8'h00, FG_G = 8'h00, FG_B = 8'h00;  //
black digits


    logic [7:0]  a, b, c;
// Night Transition
    logic [39:0] night_timer; // Timer to trigger night time
    logic night_time; // Flag to indicate if it's nighttime

    logic [15:0] dino_sprite_output;
    logic [15:0] dino_new_output;
    logic [9:0]  dino_sprite_addr;
    logic [10:0] dino_x = 100, dino_y = 248;
    logic [10:0] godzilla_x, godzilla_y;


    logic ducking, jumping;
    logic [15:0] duck_sprite_output, jump_sprite_output;
    logic [15:0] dino_left_output, dino_right_output;
```

```
    // Obstacles
    logic [10:0] s_cac_x = 1200, s_cac_y = 248;
    logic [10:0] group_x = 1600, group_y = 248;
    logic [10:0] lava_x   = 1800, lava_y   = 248;
    logic [10:0] ptr_x    = 1400, ptr_y    = 200;
    logic [10:0] powerup_x, powerup_y;

    logic [10:0] cg_x, cg_y;


    logic [15:0] scac_sprite_output, group_output, lava_output;
    logic [15:0] ptr_up_output, ptr_down_output, ptr_sprite_output;
    logic [9:0]  scac_sprite_addr, lava_sprite_addr, ptr_sprite_addr;
    logic [20:0] group_addr;
    logic [15:0] powerup_sprite_output;
    logic [9:0] powerup_sprite_addr;

  logic [15:0] godzilla_sprite_output;
  logic [9:0] godzilla_sprite_addr;
    // === Replay ===
    logic [15:0] replay_output;
    logic [9:0]  replay_addr;
    logic [10:0] replay_x = 560, replay_y = 200;

    // Motion
    logic [23:0] motion_timer;
    logic [10:0] obstacle_speed = 1;
    logic [4:0]  passed_count;
    logic        game_over;
   // logic [1:0]  sprite_state;

  // Power-up (Godzilla mode)
logic godzilla_mode;

    logic [39:0] godzilla_timer;
  //lfsr logic for random offset (obstacle positions)
    logic [5:0] lfsr;
    always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
            lfsr <= 6'b101011; // non-zero seed
        end else if (!game_over && motion_timer >= 24'd2_000_000)
begin
            // x^6 + x^5 + 1 polynomial
            lfsr <= { lfsr[4:0], lfsr[5] ^ lfsr[4] };
```

```
    end
end

function automatic logic is_visible(input logic [15:0] px);
    return (px != 16'hF81F && px != 16'hFFFF);
endfunction

function automatic logic collide(
    input logic [10:0] ax, ay, bx, by,
    input logic [5:0]  aw, ah, bw, bh
);
    return ((ax < bx + bw) && (ax + aw > bx) &&
            (ay < by + bh) && (ay + ah > by));
endfunction

always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        s_cac_x        <= 1200;
        group_x        <= 1600;
        lava_x         <= 1800;
        ptr_x          <= 1400;
        obstacle_speed <= 1;
        passed_count   <= 0;
        game_over      <= 0;
        sprite_state   <= 0;
        motion_timer   <= 0;
        score          <= 0;
        frame_counter <= 0;

        cloud_counter <= 0;
        cloud_offset <= 0;
        sky_counter <= 0;
        sky_phase <= 0;
        sky_r <= 8'd135;
        sky_g <= 8'd206;
        sky_b <= 8'd235;
        sun_counter <= 0;
        sun_offset_x <= 0;
        sun_offset_y <= 0;
        night_timer <= 32'd0;
        night_time <= 0; // Start with day
        sky_r <= 8'd135;
        sky_g <= 8'd206;
        sky_b <= 8'd235; // Day sky color
```

```
            score <= 17'd0;
            // Power-up reset
            powerup_x      <= 800;
            powerup_y      <= 248;
            godzilla_mode  <= 0;
            godzilla_timer <= 0;

        end else if (chipselect && write) begin
            case (address)
                9'd0: dino_x <= writedata[9:0];
                9'd1: dino_y <= writedata[9:0];

                9'd13: ducking <= writedata[0];
                9'd14: jumping <= writedata[0];

                9'd17: lava_x <= writedata[9:0];
                9'd18: lava_y <= writedata[9:0];
                9'd19: replay_button <= writedata[0]; // trigger
replay

            endcase

        end else if (!game_over) begin
            if (motion_timer >= 24'd2_000_000) begin
                // wrap each obstacle with a different pseudo-random
offset
                s_cac_x <= (s_cac_x <= obstacle_speed)
                           ? (HACTIVE + {lfsr,       4'd0})
                           : s_cac_x - obstacle_speed;
                group_x <= (group_x <= obstacle_speed)
                           ? (HACTIVE + {lfsr ^ 6'h3F,4'd0})
                           : group_x - obstacle_speed;
                lava_x  <= (lava_x  <= obstacle_speed)
                           ? (HACTIVE + {{lfsr[3:0]},6'd0})
                           : lava_x  - obstacle_speed;
                ptr_x   <= (ptr_x   <= obstacle_speed)
                           ? (HACTIVE + {{lfsr[5:2]},6'd0})
                           : ptr_x   - obstacle_speed;
                // Power-up movement
powerup_x <= (powerup_x <= obstacle_speed)
             ? (HACTIVE + {{lfsr[4:0]}, 5'd0})
             : powerup_x - obstacle_speed;
```

```
                bcd[0] <= bcd[0] + 1;
              for (int i = 0; i < N_DIGITS-1; i++) begin
                if (bcd[i] == 4'd10) begin
                  bcd[i]   <= 4'd0;
                  bcd[i+1] <= bcd[i+1] + 1;
                end
              end
              // wrap highest digit
              if (bcd[N_DIGITS-1] == 4'd10)
                bcd[N_DIGITS-1] <= 4'd0;
               // tick the score (wrap from 999 back to 0)
score <= (score == 17'd99999) ? 17'd0 : score + 1;                //
count passes and speed up
               if (s_cac_x <= obstacle_speed || group_x <=
obstacle_speed ||
                   lava_x  <= obstacle_speed || ptr_x   <=
obstacle_speed) begin
                   passed_count <= passed_count + 1;
               end
               if (passed_count >= 12) begin
                   obstacle_speed <= obstacle_speed + 1;
                   passed_count   <= 0;
               end

               motion_timer <= 0;
               sprite_state <= sprite_state + 1;
           end else begin
               motion_timer <= motion_timer + 1;
           end

           if (!godzilla_mode &&(collide(dino_x, dino_y, s_cac_x,
s_cac_y,  32,32,32,32) ||
               collide(dino_x, dino_y, group_x,  group_y,
64,32,32,32) ||
               collide(dino_x, dino_y, lava_x,   lava_y,
32,32,32,32) ||
                                  collide(dino_x, dino_y, ptr_x,
ptr_y,    32,32,32,32))) begin
               game_over <= 1;


           end

           if (frame_counter == 24'd5_000_000) begin
               sprite_state <= sprite_state + 1;
```

```
            frame_counter <= 0;
        end else begin
            frame_counter <= frame_counter + 1;
        end

        // Cloud drifting
        if (cloud_counter == 24'd8_000_000) begin
            cloud_counter <= 0;
            cloud_offset <= cloud_offset + 1;
            if (cloud_offset > 1280) cloud_offset <= 0;
        end else begin
            cloud_counter <= cloud_counter + 1;
        end




    // Night Timer Logic
        if (night_timer < 40'd1_500_000_000) begin
        night_timer <= night_timer + 1;  // Increment the timer
    end else if (night_timer == 40'd1_500_000_000) begin
        night_time <= ~night_time;
         night_timer <= 32'd0;
    end
        if (night_time) begin
            sky_r <= 8'd10;  // Dark blue night sky
            sky_g <= 8'd10;
            sky_b <= 8'd40;

        sun_r <= 8'd255; // White moon
        sun_g <= 8'd255;
        sun_b <= 8'd255;
        end else begin
            sky_r <= 8'd135;
            sky_g <= 8'd206;
            sky_b <= 8'd235; // Day sky color
            sun_r <= 8'd255; // Yellow sun
            sun_g <= 8'd255;
            sun_b <= 8'd0;
        end

        if (collide(dino_x, dino_y, powerup_x, powerup_y, 32, 32,
32, 32)) begin
    godzilla_mode <= 1;
```

```
    godzilla_timer <= 0;
    powerup_x <= 2000; // move off screen
end
            //Godzilla destroys
if (godzilla_mode) begin
    if (collide(dino_x, dino_y, s_cac_x, s_cac_y, 32, 32, 32, 32))
        s_cac_x <= 2000;
    if (collide(dino_x, dino_y, group_x, group_y, 64, 32, 32, 32))
        group_x <= 2000;
    if (collide(dino_x, dino_y, lava_x, lava_y, 32, 32, 32, 32))
        lava_x <= 2000;
    if (collide(dino_x, dino_y, ptr_x, ptr_y, 32, 32, 32, 32))
        ptr_x <= 2000;
end




//Godzilla timeout
if (godzilla_mode)
    godzilla_timer <= godzilla_timer + 1;

if (godzilla_timer >= 32'd100_000_000_000) begin
    godzilla_mode <= 0;
    godzilla_timer <= 0;
end
        end else begin
            // on replay, reset everything
             if (replay_button) begin
                s_cac_x        <= 1200;
                group_x        <= 1600;
                lava_x         <= 1800;
                ptr_x          <= 1400;
                obstacle_speed <= 1;
                passed_count   <= 0;
                game_over      <= 0;
                score          <= 0;
                motion_timer   <= 0;
                godzilla_mode  <= 0;
                godzilla_timer <= 0;
                powerup_x      <= 800;
                powerup_y      <= 248;
                for (int i = 0; i < N_DIGITS; i++) begin
                bcd[i] <= 4'd0;
```

```
                end


            end
        end
    end

    // VGA COUNTERS
    vga_counters counters(
        .clk50     (clk),
        .reset     (reset),
        .hcount    (hcount),
        .vcount    (vcount),
        .VGA_CLK   (VGA_CLK),
        .VGA_HS    (VGA_HS),
        .VGA_VS    (VGA_VS),
        .VGA_BLANK_n(VGA_BLANK_n),
        .VGA_SYNC_n(VGA_SYNC_n)
    );

    // SPRITE ROMS
    dino_s_cac_rom       s_cac_rom(.clk(clk),
.address(scac_sprite_addr), .data(scac_sprite_output));
    dino_cac_tog_rom     cacti_group_rom(.clk(clk),
.address(group_addr),      .data(group_output));
    dino_lava_rom        lava_rom(.clk(clk),
.address(lava_sprite_addr),  .data(lava_output));
    dino_pterodactyl_down_rom  ptero_up(.clk(clk),
.address(ptr_sprite_addr), .data(ptr_up_output));
    dino_pterodactyl_up_rom    ptero_down(.clk(clk),
.address(ptr_sprite_addr), .data(ptr_down_output));




   always_comb begin
    if (godzilla_mode)
        dino_sprite_output = godzilla_sprite_output;
    else if (ducking)
        dino_sprite_output = duck_sprite_output;
    else if (jumping)
        dino_sprite_output = jump_sprite_output;
    else begin
        case (sprite_state)
            2'd0: dino_sprite_output = dino_new_output;
```

```
            2'd1: dino_sprite_output = dino_left_output;
            2'd2: dino_sprite_output = dino_right_output;
            default: dino_sprite_output = dino_new_output;
        endcase
    end
end


    dino_sprite_rom dino_rom(.clk(clk), .address(dino_sprite_addr),
.data(dino_new_output));

    dino_duck_rom duck_rom(.clk(clk), .address(dino_sprite_addr),
.data(duck_sprite_output));
    dino_jump_rom jump_rom(.clk(clk), .address(dino_sprite_addr),
.data(jump_sprite_output));
    dino_left_leg_up_rom dino_rom1(.clk(clk),
.address(dino_sprite_addr), .data(dino_left_output));
    dino_right_leg_up_rom dino_rom2(.clk(clk),
.address(dino_sprite_addr), .data(dino_right_output));



    dino_replay_rom replay_rom(.clk(clk), .address(replay_addr),
.data(replay_output));

      dino_powerup_rom powerup_rom(.clk(clk),
.address(powerup_sprite_addr), .data(powerup_sprite_output));

      dino_godzilla_rom godzilla_rom(.clk(clk),
.address(godzilla_sprite_addr), .data(godzilla_sprite_output));



    // Pterodactyl animation
    always_comb begin
        case (sprite_state)
            2'd0: ptr_sprite_output = ptr_up_output;
            2'd1: ptr_sprite_output = ptr_down_output;
            default: ptr_sprite_output = ptr_up_output;
        endcase
    end
```

```
always_ff @(posedge clk) begin
    a <= 8'd135; b <= 8'd206; c <= 8'd235;

    if (sample_clock >= 285) begin  // 50MHz / 175550 ≈ 285
    sample_clock <= 0;
    audio_sample <= audio_data[audio_index];

    if (audio_index == 9659)
        audio_index <= 0;
    else
        audio_index <= audio_index + 1;
end else begin
    sample_clock <= sample_clock + 1;
end

if (L_READY) begin
    L_DATA  <= audio_sample;
    L_VALID <= (sample_clock == 0);
end else begin
    L_VALID <= 0;
end

if (R_READY) begin
    R_DATA  <= audio_sample;
    R_VALID <= (sample_clock == 0);
end else begin
    R_VALID <= 0;
end




    if (!game_over ) begin
//begin



    if (vcount < 280) begin
        a <= sky_r;
        b <= sky_g;
        c <= sky_b;
    end else if (vcount > 300) begin
        a <= 8'd100;
        b <= 8'd40;
```

```
        c <= 8'd10;
    end else begin
        a <= 8'd139;
        b <= 8'd69;
        c <= 8'd19;

    end

    //Ground Line
    if (vcount == 280) begin
        a <= 8'd0;
        b <= 8'd0;
        c <= 8'd0;
    end


    if ((hcount-(1150-sun_offset_x))*(hcount-(1150-sun_offset_x)) +
        (vcount-(80+sun_offset_y))*(vcount-(80+sun_offset_y)) < 1200
&&
        (hcount-(1150-sun_offset_x))*(hcount-(1150-sun_offset_x)) +
        (vcount-(80+sun_offset_y))*(vcount-(80+sun_offset_y)) > 900)
begin
        a <= sun_r;
        b <= sun_g;
        c <= sun_b;
    end
     if ((hcount-(1150-sun_offset_x))*(hcount-(1150-sun_offset_x)) +
        (vcount-(80+sun_offset_y))*(vcount-(80+sun_offset_y)) < 900)
begin
        a <= sun_r;
        b <= sun_g;
        c <= sun_b;
    end

    //
    // --- Cloud 1 ---
    if (((hcount-(235+cloud_offset))*(hcount-(235+cloud_offset)) +
(vcount-70)*(vcount-70) < 100) ||
        ((hcount-(245+cloud_offset))*(hcount-(245+cloud_offset)) +
(vcount-65)*(vcount-65) < 100) ||
        ((hcount-(255+cloud_offset))*(hcount-(255+cloud_offset)) +
(vcount-65)*(vcount-65) < 100) ||
        ((hcount-(245+cloud_offset))*(hcount-(245+cloud_offset)) +
(vcount-75)*(vcount-75) < 100) ||
```

```
        ((hcount-(255+cloud_offset))*(hcount-(255+cloud_offset)) +
(vcount-75)*(vcount-75) < 100) ||
        ((hcount-(265+cloud_offset))*(hcount-(265+cloud_offset)) +
(vcount-70)*(vcount-70) < 100) ||


((hcount-(235+cloud_offset-1280))*(hcount-(235+cloud_offset-1280)) +
(vcount-70)*(vcount-70) < 100) ||

((hcount-(245+cloud_offset-1280))*(hcount-(245+cloud_offset-1280)) +
(vcount-65)*(vcount-65) < 100) ||

((hcount-(255+cloud_offset-1280))*(hcount-(255+cloud_offset-1280)) +
(vcount-65)*(vcount-65) < 100) ||

((hcount-(245+cloud_offset-1280))*(hcount-(245+cloud_offset-1280)) +
(vcount-75)*(vcount-75) < 100) ||

((hcount-(255+cloud_offset-1280))*(hcount-(255+cloud_offset-1280)) +
(vcount-75)*(vcount-75) < 100) ||

((hcount-(265+cloud_offset-1280))*(hcount-(265+cloud_offset-1280)) +
(vcount-70)*(vcount-70) < 100)) begin
        a <= 8'd255;
        b <= 8'd255;
        c <= 8'd255;
    end
        // --- Cloud 2 ---
    if (((hcount-(440+cloud_offset))*(hcount-(440+cloud_offset)) +
(vcount-100)*(vcount-100) < 100) ||
        ((hcount-(450+cloud_offset))*(hcount-(450+cloud_offset)) +
(vcount-95)*(vcount-95) < 100) ||
        ((hcount-(460+cloud_offset))*(hcount-(460+cloud_offset)) +
(vcount-95)*(vcount-95) < 100) ||
        ((hcount-(440+cloud_offset))*(hcount-(440+cloud_offset)) +
(vcount-105)*(vcount-105) < 100) ||
        ((hcount-(450+cloud_offset))*(hcount-(450+cloud_offset)) +
(vcount-110)*(vcount-110) < 100) ||
        ((hcount-(460+cloud_offset))*(hcount-(460+cloud_offset)) +
(vcount-105)*(vcount-105) < 100) ||


((hcount-(440+cloud_offset-1280))*(hcount-(440+cloud_offset-1280)) +
(vcount-100)*(vcount-100) < 100) ||
```

```
((hcount-(450+cloud_offset-1280))*(hcount-(450+cloud_offset-1280)) +
(vcount-95)*(vcount-95) < 100) ||

((hcount-(460+cloud_offset-1280))*(hcount-(460+cloud_offset-1280)) +
(vcount-95)*(vcount-95) < 100) ||

((hcount-(440+cloud_offset-1280))*(hcount-(440+cloud_offset-1280)) +
(vcount-105)*(vcount-105) < 100) ||

((hcount-(450+cloud_offset-1280))*(hcount-(450+cloud_offset-1280)) +
(vcount-110)*(vcount-110) < 100) ||

((hcount-(460+cloud_offset-1280))*(hcount-(460+cloud_offset-1280)) +
(vcount-105)*(vcount-105) < 100)) begin
        a <= 8'd250;
        b <= 8'd250;
        c <= 8'd250;
    end
        // --- Cloud 3 ---
    if (((hcount-(690+cloud_offset))*(hcount-(690+cloud_offset)) +
(vcount-60)*(vcount-60) < 100) ||
        ((hcount-(700+cloud_offset))*(hcount-(700+cloud_offset)) +
(vcount-55)*(vcount-55) < 100) ||
        ((hcount-(710+cloud_offset))*(hcount-(710+cloud_offset)) +
(vcount-55)*(vcount-55) < 100) ||
        ((hcount-(690+cloud_offset))*(hcount-(690+cloud_offset)) +
(vcount-65)*(vcount-65) < 100) ||
        ((hcount-(700+cloud_offset))*(hcount-(700+cloud_offset)) +
(vcount-70)*(vcount-70) < 100) ||
        ((hcount-(710+cloud_offset))*(hcount-(710+cloud_offset)) +
(vcount-65)*(vcount-65) < 100) ||


((hcount-(690+cloud_offset-1280))*(hcount-(690+cloud_offset-1280)) +
(vcount-60)*(vcount-60) < 100) ||

((hcount-(700+cloud_offset-1280))*(hcount-(700+cloud_offset-1280)) +
(vcount-55)*(vcount-55) < 100) ||

((hcount-(710+cloud_offset-1280))*(hcount-(710+cloud_offset-1280)) +
(vcount-55)*(vcount-55) < 100) ||

((hcount-(690+cloud_offset-1280))*(hcount-(690+cloud_offset-1280)) +
(vcount-65)*(vcount-65) < 100) ||
```

```
((hcount-(700+cloud_offset-1280))*(hcount-(700+cloud_offset-1280)) +
(vcount-70)*(vcount-70) < 100) ||

((hcount-(710+cloud_offset-1280))*(hcount-(710+cloud_offset-1280)) +
(vcount-65)*(vcount-65) < 100)) begin
        a <= 8'd245;
        b <= 8'd245;
        c <= 8'd245;
    end

    // Tiny Birds
    if (((hcount > 300 && hcount < 305) && (vcount == 50)) ||
        ((hcount > 305 && hcount < 310) && (vcount == 51)) ||
        ((hcount > 310 && hcount < 315) && (vcount == 50)) ||
        ((hcount > 600 && hcount < 605) && (vcount == 80)) ||
        ((hcount > 605 && hcount < 610) && (vcount == 81)) ||
        ((hcount > 610 && hcount < 615) && (vcount == 80))) begin
        a <= 8'd0;
        b <= 8'd0;
        c <= 8'd0;
    end
         //  Ground Rocks
    if (vcount > 280 && vcount < 480) begin
        if ((hcount % 120 == 0 && vcount % 50 < 10) ||
            (hcount % 200 == 15 && vcount % 60 < 8)) begin
            a <= 8'd110;
            b <= 8'd50;
            c <= 8'd10;
        end
    end


        // Power-up sprite drawing
if (hcount >= powerup_x && hcount < powerup_x + 32 &&
    vcount >= powerup_y && vcount < powerup_y + 32) begin
    powerup_sprite_addr <= (hcount - powerup_x) + ((vcount -
powerup_y) * 32);
    if (is_visible(powerup_sprite_output)) begin
        a <= {powerup_sprite_output[15:11], 3'b000};
        b <= {powerup_sprite_output[10:5],  2'b00};
        c <= {powerup_sprite_output[4:0],   3'b000};
    end
end
```

```
        if (hcount >= dino_x && hcount < dino_x + 32 && vcount >=
dino_y && vcount < dino_y + 32) begin
    if (godzilla_mode)
        godzilla_sprite_addr <= (hcount - dino_x) + ((vcount -
dino_y) * 32);
    else
        dino_sprite_addr <= (hcount - dino_x) + ((vcount - dino_y) *
32);

    if (is_visible(dino_sprite_output)) begin
        a <= {dino_sprite_output[15:11], 3'b000};
        b <= {dino_sprite_output[10:5],  2'b00};
        c <= {dino_sprite_output[4:0],   3'b000};
    end
end

        if (hcount >= s_cac_x && hcount < s_cac_x + 32 && vcount >=
s_cac_y && vcount < s_cac_y + 32) begin
            scac_sprite_addr <= (hcount - s_cac_x) + ((vcount -
s_cac_y) * 32);
            if (is_visible(scac_sprite_output)) begin
                a <= {scac_sprite_output[15:11], 3'b000};
                b <= {scac_sprite_output[10:5],  2'b00};
                c <= {scac_sprite_output[4:0],   3'b000};
            end
        end
        if (hcount >= group_x && hcount < group_x + 64 && vcount >=
group_y && vcount < group_y + 32) begin
            group_addr <= (hcount - group_x) + ((vcount - group_y) *
64);
            if (is_visible(group_output)) begin
                a <= {group_output[15:11], 3'b000};
                b <= {group_output[10:5],  2'b00};
                c <= {group_output[4:0],   3'b000};
            end
        end
        if (hcount >= lava_x && hcount < lava_x + 32 && vcount >=
lava_y && vcount < lava_y + 32) begin
            lava_sprite_addr <= (hcount - lava_x) + ((vcount -
lava_y) * 32);
            if (is_visible(lava_output)) begin
                a <= {lava_output[15:11], 3'b000};
                b <= {lava_output[10:5],  2'b00};
                c <= {lava_output[4:0],   3'b000};
            end
```

```
        end
        if (hcount >= ptr_x && hcount < ptr_x + 32 && vcount >= ptr_y
&& vcount < ptr_y + 32) begin
            ptr_sprite_addr <= (31 - (hcount - ptr_x)) + ((vcount -
ptr_y) * 32);
            if (is_visible(ptr_sprite_output)) begin
                a <= {ptr_sprite_output[15:11], 3'b000};
                b <= {ptr_sprite_output[10:5],  2'b00};
                c <= {ptr_sprite_output[4:0],   3'b000};
            end
        end

      if (vcount >= SCORE_Y && vcount < SCORE_Y + 8) begin
    if (hcount >= SCORE_X && hcount < (SCORE_X + N_DIGITS * 8)) begin
        rx  = hcount - SCORE_X;
        idx = rx / 8;          // Each digit is 8 pixels wide
        cx  = rx % 8;
        ry  = vcount - SCORE_Y;

        if (idx < N_DIGITS && cx < 8) begin
            if (font_rom[bcd[N_DIGITS - 1 - idx]][ry][7 - cx]) begin
                a <= FG_R;
                b <= FG_G;
                c <= FG_B;
            end
        end
    end
end


    end else begin
        if (hcount >= replay_x && hcount < replay_x + 160 && vcount
>= replay_y && vcount < replay_y + 32) begin
            replay_addr <= (hcount - replay_x) + ((vcount - replay_y)
* 160);
            if (is_visible(replay_output)) begin
                a <= {replay_output[15:11], 3'b000};
                b <= {replay_output[10:5],  2'b00};
                c <= {replay_output[4:0],   3'b000};
            end
        end

    end

end
```

```
assign {VGA_R, VGA_G, VGA_B} = {a, b, c};

endmodule

 module vga_counters(
    input  logic        clk50, reset,
    output logic [10:0] hcount,
    output logic [9:0]  vcount,
    output logic        VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
VGA_SYNC_n
);

   parameter HACTIVE = 11'd1280,
             HFRONT = 11'd32,
             HSYNC  = 11'd192,
             HBACK  = 11'd96,
             HTOTAL = HACTIVE + HFRONT + HSYNC + HBACK;

   parameter VACTIVE = 10'd480,
             VFRONT = 10'd10,
             VSYNC  = 10'd2,
             VBACK  = 10'd33,
             VTOTAL = VACTIVE + VFRONT + VSYNC + VBACK;

   logic endOfLine;
   always_ff @(posedge clk50 or posedge reset)
      if (reset)
         hcount <= 0;
      else if (endOfLine)
         hcount <= 0;
      else
         hcount <= hcount + 1;

   assign endOfLine = (hcount == HTOTAL - 1);

   logic endOfField;
   always_ff @(posedge clk50 or posedge reset)
      if (reset)
         vcount <= 0;
      else if (endOfLine)
         if (endOfField)
            vcount <= 0;
         else
            vcount <= vcount + 1;
```

```
   assign endOfField = (vcount == VTOTAL - 1);

   assign VGA_HS = !((hcount >= (HACTIVE + HFRONT)) && (hcount <
(HACTIVE + HFRONT + HSYNC)));
   assign VGA_VS = !((vcount >= (VACTIVE + VFRONT)) && (vcount <
(VACTIVE + VFRONT + VSYNC)));
   assign VGA_SYNC_n = 1'b0;
   assign VGA_BLANK_n = (hcount < HACTIVE) && (vcount < VACTIVE);
   assign VGA_CLK = hcount[0];

   endmodule
```

dinofinal.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <stdbool.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <libusb-1.0/libusb.h>
#include "usbkeyboard.h"

#define REPORT_LEN         8
#define LW_BRIDGE_BASE     0xFF200000
#define MAP_SIZE           0x1000

#define DINO_Y_OFFSET      (1 * 4)
#define DUCKING_OFFSET     (13 * 4)
#define JUMPING_OFFSET     (14 * 4)
#define REPLAY_OFFSET      (19 * 4)

#define GROUND_Y           248
#define FIXED_SHIFT        4                    // 1 pixel = 16 units
#define GROUND_Y_FIXED     (GROUND_Y << FIXED_SHIFT)

#define INITIAL_VELOCITY   (-84)                // -4.0 in fixed-point
(higher jump)
#define GRAVITY            1                    // 1/16th pixel per
loop
#define DELAY_US           5000
```

```
int main(void) {
    int fd = open("/dev/mem", O_RDWR | O_SYNC);
    if (fd < 0) { perror("open(/dev/mem)"); return 1; }
    void *lw_base = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, LW_BRIDGE_BASE);
    if (lw_base == MAP_FAILED) { perror("mmap"); return 1; }

    volatile uint32_t *dino_y_reg = (uint32_t *)(lw_base +
DINO_Y_OFFSET);
    volatile uint32_t *duck_reg   = (uint32_t *)(lw_base +
DUCKING_OFFSET);
    volatile uint32_t *jump_reg   = (uint32_t *)(lw_base +
JUMPING_OFFSET);
    volatile uint32_t *replay_reg = (uint32_t *)(lw_base +
REPLAY_OFFSET);

    struct libusb_device_handle *pad;
    uint8_t ep;
    pad = openkeyboard(&ep);
    if (!pad) {
        fprintf(stderr, "Controller not found\n");
        munmap(lw_base, MAP_SIZE);
        close(fd);
        return 1;
    }

    int y_fixed = GROUND_Y_FIXED;
    int v_fixed = 0;

    unsigned char report[REPORT_LEN];
    int transferred, r;

    while (1) {
        r = libusb_interrupt_transfer(pad, ep, report, REPORT_LEN,
&transferred, 0);
        if (r < 0) {
            fprintf(stderr, "USB read error: %d\n", r);
            break;
        }

        uint8_t y_axis = report[4];
        bool want_jump = (y_axis == 0x00 && y_fixed ==
GROUND_Y_FIXED);
        bool want_duck = (y_axis == 0xFF && y_fixed ==
GROUND_Y_FIXED);
```

```
        bool want_replay = (report[6] & 0x20);

        if (want_jump) v_fixed = INITIAL_VELOCITY;

        v_fixed += GRAVITY;
        y_fixed += v_fixed;

        if (y_fixed > GROUND_Y_FIXED) {
            y_fixed = GROUND_Y_FIXED;
            v_fixed = 0;
        }

        *dino_y_reg = y_fixed >> FIXED_SHIFT;
        *jump_reg = want_jump;
        *duck_reg = want_duck;
        *replay_reg = want_replay;

        usleep(DELAY_US);
    }

    libusb_close(pad);
    libusb_exit(NULL);
    munmap(lw_base, MAP_SIZE);
    close(fd);
    return 0;
}
```