CSEE 4840 Embedded System

Flappy Bird Final Report

Ethan Yang

Tianshuo Jin

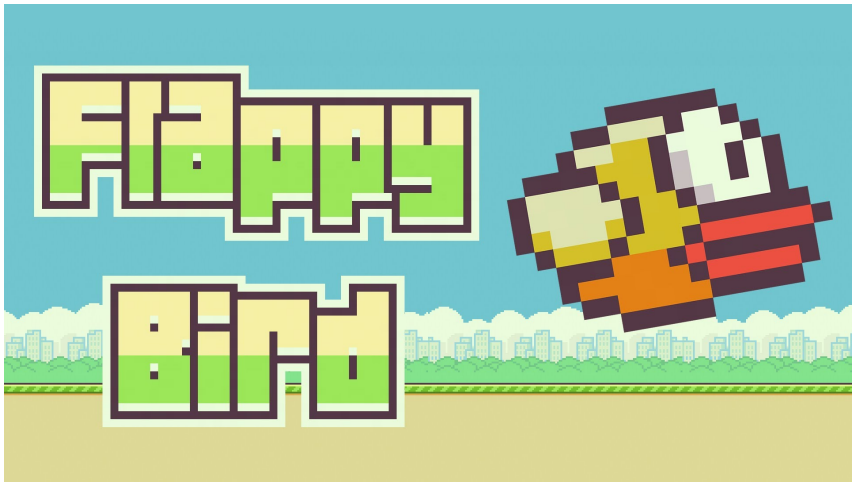
Zidong Xu

Sijun Li

Contents

1 Introduction

We are implementing an enhanced version of the popular game Flappy Bird on the DE1-SoC board. This side-scrolling game has the player controlling a bird trying to fly between rows of green pipes without hitting them. Each time the player safely passes through a pillar without hitting it, they score 1 point; if they hit a pillar, the game is over. Each time the player presses a button, the bird briefly jumps upwards; if the button is not pressed, the bird falls due to gravity. For the game to flow naturally, we need an infinitely scrolling background simulation and pixel-perfect sprite-based collision detection. This project aims to implement Flappy Bird on the DE1-SoC board using hardware-accelerated graphics rendering and software-driven game logic. To do this, you need to implement graphics rendering, such as physics simulation for gravity and collisions, keyboard input for player control, and a scoring system.



2 Overview

This project implements a hardware version of the popular game *Flappy Bird* on the DE1-SoC FPGA development board. The goal is to create a playable, responsive game that demonstrates the synergy between software and digital hardware design. The game is rendered via a custom VGA controller written in Verilog and is controlled using a USB keyboard, with user input processed in C and transmitted to the hardware through a Linux kernel module.

Unlike traditional software-only implementations, this version of Flappy Bird places the entire game logic in hardware. The core gameplay — including bird movement, pipe generation, gravity simulation, collision detection, scoring, and game state transitions — is entirely implemented in Verilog (`vga_ball.sv`). The bird flaps upward when a flap signal is received, which is triggered by the spacebar key on a connected USB keyboard.

To enable communication between the software and hardware layers, a custom Linux device driver (`vga_ball.c`) exposes memory-mapped registers via `/dev/vga_ball`. These regis-

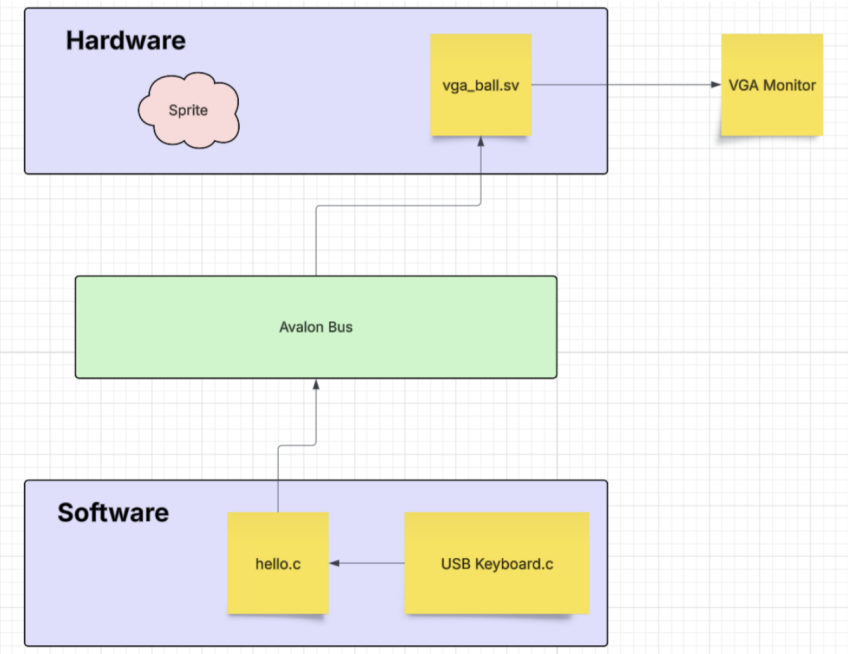
ters allow userspace programs to trigger events (such as a flap) or initialize values (such as bird position and background color) using `ioctl()` system calls.

A lightweight C program (`hello.c`) handles USB input using the `libusb` library. It detects keypresses from the USB keyboard and, upon detecting a spacebar press, sends a flap command to the FPGA by writing to the appropriate register. The `ESC` key is used to terminate the game.

The result is a responsive, fully playable Flappy Bird game that runs entirely on FPGA hardware, controlled through real-time keyboard input, and rendered directly to a VGA display.

3 System Architecture

3.1 Block Diagram



3.2 Hardware Components

- **VGA Controller Module**: Generates VGA signals (640×480@60Hz) to display the game.
- **Game Logic Module**: Implements game physics, collision detection, and state management.
- **Graphics Renderer**: Handles multi-layered sprite rendering and animation.
- **Memory-Mapped Interface**: Provides software control through register access.

3.3 Software Components

- **Linux Device Driver**: Interfaces with the hardware through memory-mapped registers.
- **User Application**: Processes keyboard input and sends commands to hardware.
- **USB Keyboard Interface**: Captures spacebar presses for flap actions.

3.4 Memory-Mapped Register Table

Offset	Function	Description
0–2	Background color	R, G, B components (0–255)
3–4	X position (not in use)	Low 8 bits (3), High 2 bits (4)
5–6	Y position (not in use)	Low 8 bits (5), High 2 bits (6)
7	Flap signal (replaced radius)	Input from keyboard (0/1)

3.5 Internal Register Table

Register	Width	Purpose
`bird_y`	[9:0]	Vertical position of the bird
`bird_velocity`	signed [9:0]	Vertical speed; affected by gravity and flap
`bird_frame`	[1:0]	Index of bird animation frame (0–2)
`animation_counter`	[7:0]	Counter to cycle bird frames
`vsync_reg`	1	Stores VGA_VS for edge detection
`game_started`	1	0 = idle animation; 1 = game running
`flap_latched`	1	Latched flap input, reset once consumed
`bird_y_idle`	[9:0]	Bird idle animation Y position
`idle_dir`	1	Direction of idle movement (0 = up, 1 = down)
`hcount`, `vcount`	[10:0], [9:0]	VGA counters for pixel row/column
`bg_addr`	[18:0]	Address into background ROM
`bg_color`	[7:0]	Pixel color read from background ROM
`bird_addr`	[11:0]	Address into current bird sprite ROM
`bird_color`	[7:0]	Selected pixel color from current frame ROM
`pipe_x[i]`	[9:0] (array)	X position of each pipe pair (3 total)
`pipe_gap_y[i]`	[9:0] (array)	Y position of the top of the gap in each pipe pair
`bird_color0/1/2`	[7:0]	Outputs of 3 sprite ROMs for animation

4 Module Implementation

4.1 Sprite Controller

The image processing module is responsible for converting static game assets (e.g., background, pipes, birds, and overlays like the game over screen) into memory-initialized ROM files suitable for use within the FPGA design. These image assets are preprocessed offline using a custom Python script and converted into `.mif` (Memory Initialization File) format.

Each image is transformed into a single-port ROM memory block with 8-bit indexed color representation. The images are first quantized and palette-indexed, and then their pixel data is serialized into `.mif` format that defines memory width, depth, and pixel values in hexadecimal, enabling block memory instantiation in Verilog.

The Python script `convert.py` automates this image conversion pipeline. It accepts `.png` input images and generates corresponding `.mif` files that can be directly used in the Verilog design. For example, `Gameover.png` is converted into `Gameover.mif`, which stores the pixel data that is displayed when the game ends.

All game graphics—including `bg`, `base`, `bird0`, `bird1`, `bird2`, and `gameover`—are handled using this approach. Each ROM module loads a specific image's `.mif` file and provides addressable pixel data during rendering.

By isolating static texture data in ROMs, the image processing module allows the VGA and sprite controller logic to focus purely on logical state updates and object positioning, rather than managing individual pixel values. This significantly simplifies real-time rendering logic on the hardware side and enables a modular, scalable approach to image asset integration.

Sprite	Number	Bit width	Depth	ROM Size	Example
base	1	8	37632	301056	
bg	1	8	307200	2457600	
Bird0, 1, 2	3	8	816	19584	
Game over	1	8	8064	64512	

4.2 VGA Controller

The VGA controller is the foundation of our Flappy Bird implementation, responsible for generating precise timing signals that drive the 640×480@60Hz display. This module

handles both the signal generation and the sophisticated sprite rendering system that brings the game to life.

VGA Signal Generation

The controller uses two counters to generate the VGA timing signals:

- `hcount`: 11-bit counter (0 to 1599), tracking the horizontal position.
- `vcount`: 10-bit counter (0 to 524), tracking the vertical position.

The counters increment on each clock cycle (50 MHz), with the horizontal counter resetting at the end of each line and triggering an increment of the vertical counter. Since the VGA standard requires a 25 MHz pixel clock, we effectively display one pixel every two clock cycles, using `hcount[10:1]` as the pixel column address.

Multi-Layered Sprite System

Our Flappy Bird implementation features a sophisticated sprite rendering system that manages multiple graphical elements.

All game sprites are stored in separate ROM modules:

```
bg_rom    bg_rom_inst     (.address(bg_addr), .clock(clk), .q(bg_color));
bird_rom0 bird0           (.address(bird_addr), .clock(clk), .q(bird_color0));
bird_rom1 bird1           (.address(bird_addr), .clock(clk), .q(bird_color1));
bird_rom2 bird2           (.address(bird_addr), .clock(clk), .q(bird_color2));
gameover_rom gameover_inst(.address(gameover_addr), .clock(clk), .q(gameover_color));
base_rom  ground_inst     (.address(ground_addr), .clock(clk), .q(ground_color));
```

- Each ROM contains pre-designed pixel data with color information.
- The hardware accesses these ROMs in real-time during the rendering process.
- This approach allows for detailed graphics without consuming excessive FPGA logic resources.

Layer Priority Hierarchy

The rendering system implements a strict priority hierarchy to ensure correct visual composition. The priority, from highest to lowest, is as follows:

1. Score display (highest priority)
2. Game over message (only in `GAME_OVER` state)
3. Bird sprite with animation
4. Pipes (simple colored rectangles)

5. Ground layer with texture
6. Background (sky) – lowest priority

Higher priority layers override lower ones when they overlap. This hierarchy ensures that critical game elements remain clearly visible at all times.

Address Calculation for Sprites

Each sprite has specific dimensions that define its size on screen. For example, the bird sprite is 34×24 pixels, and its ROM address is calculated using the following formula:

$$\texttt{bird\_addr} = (\texttt{vcount} - \texttt{bird\_y}) \times \texttt{BIRD\_WIDTH} + (\texttt{hcount[10:1]} - \texttt{BIRD\_X})$$

Similar calculations are used for background, ground, and the game over screen:

$$\texttt{bg\_addr} = \texttt{vcount} \times 640 + ((\texttt{hcount[10:1]} + \texttt{scroll\_offset}) \bmod 640)$$

$$\texttt{ground\_addr} = (\texttt{vcount} - 440) \times 640 + ((\texttt{hcount[10:1]} + \texttt{scroll\_offset}) \bmod 640)$$

$$\texttt{gameover\_addr} = (\texttt{vcount} - \texttt{GAMEOVER\_Y}) \times \texttt{GAMEOVER\_WIDTH} + (\texttt{hcount[10:1]} - \texttt{GAMEOVER\_X})$$

These calculations effectively map from screen coordinates to corresponding ROM memory addresses. The combinational logic pipeline ensures that these addresses are computed in advance, allowing pixel data to be accessed on time for each frame.

Transparency and Color Management

Sprite Transparency: Transparent pixels allow for irregular sprite shapes. A special color value `0x00` is used to indicate transparency. The rendering logic checks the pixel value before displaying it:

```
if (bird_color_reg != 8'h00) begin
    // Render bird pixel
end else begin
    // Render next lower priority layer
end
```

This technique enables smooth overlays and is especially important for the bird sprite, which does not occupy a full rectangular region.

Color Conversion: The color data in ROM is stored in an efficient 8-bit `RGB332` format (3 bits for red, 3 bits for green, 2 bits for blue). Hardware expands this to 24-bit RGB by padding each channel:

```
VGA_R = {bird_color_reg[7:5], 5'b00000};
VGA_G = {bird_color_reg[4:2], 5'b00000};
VGA_B = {bird_color_reg[1:0], 6'b000000};
```

This conversion maintains good color fidelity while optimizing memory usage. The asymmetrical bit allocation (3-3-2) corresponds to the human eye's greater sensitivity to red and green than to blue.

4.3 Background Scrolling Module

In this design, the background scrolling is driven by a synchronization register `scroll_offset` and a counter `scroll_counter`. The `scroll_counter` is incremented on each clock cycle, and when it reaches a preset threshold, `scroll_offset` is increased by one and wrapped around using modulo 640 (the width of the background in pixels). After that, `scroll_counter` is reset to zero.

During the pixel rendering stage, the column coordinate of each pixel (`hcount[10:1]`) is added to the current `scroll_offset`, and the result is taken modulo 640 to determine the actual column index of the background image. The row index is calculated using `vcount*640`. This method ensures that the background offset remains constant within a single frame, without the need to physically shift the entire image or use a frame buffer.

Instead, simple combinational logic is used to directly map the computed address to the ROM that stores the background data, achieving a smooth and efficient circular scrolling effect for the background.

4.4 Bird Logic Module

The bird logic module is responsible for implementing the vertical movement behavior of the bird, including gravity-induced falling, jump response, boundary constraints, and state updates. This module updates the bird's position and velocity once per frame, based on the current state and user input signal, and transmits the updated coordinates to both the sprite control module and the VGA controller for rendering and collision detection.

During gameplay, the bird remains at a fixed horizontal position slightly left of the center of the screen, moving only along the vertical axis. Its vertical position is represented by the register `bird_y` (in pixels, using integer precision), and its vertical velocity is stored in a signed register `bird_v` (in pixels per frame).

The module operates on a frame-based timing schedule (e.g., 60 Hz), updating both `bird_y` and `bird_v` at the start of each frame. Gravity is applied by incrementing `bird_v` with a constant value (e.g., `GRAVITY = 1`) every frame, creating a continuous falling effect. The updated position is then calculated as:

$$\texttt{bird\_y} = \texttt{bird\_y} + \texttt{bird\_v}$$

When the user presses the jump button, the module immediately sets the bird's vertical velocity to a fixed negative value (e.g., -12), simulating an upward impulse. To avoid

repeated triggering and signal noise, the input signal is processed with edge detection logic to respond only to rising edges.

Additionally, a jump cooldown mechanism is implemented (for example, allowing only one jump per 3 frames) to prevent multiple triggers caused by a long key press. This ensures smoother gameplay and prevents unnatural upward acceleration when holding the jump button.

4.5 Pipe Generation

In our implementation of Flappy Bird, the pipe system is designed to simulate a continuous stream of obstacles by reusing a fixed number of pipes. This section explains how the pipe behavior is controlled in terms of parameters, movement, rendering, gap generation, initialization, and recycling logic.

Each pipe is represented using a `pipe_t` struct, which includes two fields: `pipes[i].x` represents the x-coordinate of the left edge of the pipe, and `pipes[i].gap_y` defines the vertical position of the top of the gap. The pipe width is fixed at 52 pixels, and the height of the gap is also fixed at 100 pixels. So the passable gap lies between `gap_y` and `gap_y + GAP_HEIGHT`.

Pipe movement is achieved by continuously decreasing the value of `pipes[i].x`. When the game is in the `PLAYING` state, every time a vertical sync (`VGA_VS`) occurs, we check for a rising edge using `(VGA_VS && !vsync_reg)` and then subtract 2 pixels from each pipe's x-position. This happens once per frame, keeping the scrolling speed consistent and synchronized with the display refresh rate.

Rendering the pipes happens in a combinational block that runs every frame. The system checks if the current pixel lies within the horizontal range of any pipe (`pipes[j].x` to `pipes[j].x + PIPE_WIDTH`). If the pixel's y-coordinate is outside of the gap (either above `gap_y` or below `gap_y + GAP_HEIGHT`), the pixel is considered part of the pipe and is colored green. This creates a rectangular pipe with a vertical gap in the middle.

The gap position `gap_y` is generated using a linear feedback shift register (LFSR), which provides pseudo-random values with very low hardware cost. Every time a pipe recycles back to the right side of the screen, we trigger the LFSR and compute `gap_y` using `lfsr % 160 + 80`. This ensures the gap stays within a reasonable range, roughly between y = 80 and y = 240, so the game always remains playable.

To give the player a smoother start, the first few pipes are initialized with fixed `gap_y` values, such as `150 + i × 20`, instead of random ones. This ensures that the initial gaps are well spaced and easy to fly through, helping the player get used to the game mechanics.

The recycling logic works as follows: during each frame, we check whether a pipe has completely moved off the left side of the screen (`pipes[i].x <= 1`). If so, instead of

deleting the pipe, we reposition it to the right side by setting its x-position to the current rightmost pipe's x-coordinate plus PIPE_SPACING. To do this, we first iterate through all pipes to find the largest pipes[i].x, store it as max_pipe_x, and then set the recycled pipe's x-position to max_pipe_x + PIPE_SPACING. At the same time, we assign a new gap_y to the pipe using the LFSR method. This way, with only 3 to 5 pipes, we can simulate an endless flow of obstacles entering from the right, while keeping spacing and randomness under control.

Finally, we initialize the pipes starting from pipes[i].x = 440 + i × PIPE_SPACING, meaning the first pipe begins just outside the visible screen area. This allows the first pipe to enter the screen shortly after the game starts, so the player doesn't have to wait too long to encounter the first obstacle. This helps maintain good pacing and makes the game feel responsive from the beginning.

4.6 Collision Detection Module

In the collision detection logic, the bird is treated as a rectangle. The horizontal position (BIRD_X) represents the bird's left edge, with a fixed width of 34 pixels, and the vertical position (bird_y) represents the top edge, with a fixed height of 24 pixels. With these four parameters — BIRD_X, BIRD_X + BIRD_WIDTH, bird_y, and bird_y + BIRD_HEIGHT — we can fully define the bounding box of the bird.

For the pipes, we focus on each pipe's left edge (pipes[i].x) and its gap region. The top of the gap is given by pipes[i].gap_y, and the bottom is pipes[i].gap_y + GAP_HEIGHT. Using these values, we can determine the boundaries needed for collision detection between the bird and the pipes.

There are two main types of collisions:

The first is ground collision. The ground is set at y = 440, so we simply check whether the bottom of the bird exceeds this value. That is, if bird_y + BIRD_HEIGHT > 440, a ground collision is triggered and the game enters the GAME_OVER state.

The second is pipe collision. This is checked in two steps. First, we determine whether the bird's horizontal range overlaps with any pipe's horizontal range, using the condition:

```
BIRD_X + BIRD_WIDTH > pipes[i].x   &&   BIRD_X < pipes[i].x +
PIPE_WIDTH
```

If there is horizontal overlap, we then check whether the bird's vertical range is completely outside the gap area. This is the case when:

```
bird_y < pipes[i].gap_y   ||   bird_y + BIRD_HEIGHT > pipes[i].gap_y
+ GAP_HEIGHT
```

If both conditions are satisfied, it means the bird has collided with the pipe, and the game logic triggers a collision event.

4.7 Score Module

In the `PLAYING` state, the score is stored in a 16-bit register `score`. During each vertical synchronization cycle, the horizontal coordinate of each pipe decreases. When the right edge of a pipe crosses the bird's x-position for the first time — that is, when the following condition is met:

```
(pipes[i].x + PIPE_WIDTH < BIRD_X)   &&   (pipes[i].x + PIPE_WIDTH
>= BIRD_X - \Delta)
```

where Δ is the horizontal movement step per frame, the `score` register is incremented by 1.

In the `WAITING` state, the score is reset to 0.

For display, the 16-bit score is first split into tens and ones using combinational logic:

$$\texttt{digit0} = (\texttt{score}/10)\%10, \quad \texttt{digit1} = \texttt{score}\%10$$

Then, two sets of `case` statements are used to map each decimal digit to a 7-bit seven-segment encoding signal `{A...G}`.

During the `always_comb` block for each pixel rendering cycle, each segment of the two-digit number is drawn by calling the `in_rect(x0, y0, W, H, hcount[10:1], vcount)` function. This checks whether the current pixel lies within the rectangular area corresponding to that segment.

If the segment signal is 1 and the pixel is within its area, `score_pixel` is set to 1. Finally, when `VGA_BLANK_n` permits rendering, the `score_pixel` is drawn in white. Other pixels are rendered according to the defined priority: bird, pipe, ground, and background.

4.8 USB Keyboard Module

The USB keyboard module in this project enables interaction between the player and the FPGA-based game system. Its primary function is to capture spacebar input to trigger the bird's flap action. This module is implemented through coordination between the software and hardware layers. On the software side, the `libusb 1.0` library is used to interface with USB HID keyboards. The `openkeyboard()` function performs device enumeration, configures the interface, and retrieves the appropriate endpoint address. The application polls the keyboard every 10 milliseconds, and upon detecting a spacebar press (keycode 0x2C), it triggers a flap command. Pressing the ESC key (keycode 0x29) terminates the program.

The flap action is issued using an `ioctl()` system call, which sends a control request to the device driver. The driver's `write_flap()` function processes the request, normalizes the input to binary (0 or 1), writes the signal to memory-mapped register 7, and logs the action in the kernel. To support this mechanism, register 7 was repurposed from its original use (ball radius) to represent the flap control signal. In each frame, the hardware checks the value of this register. If the value is 1, a flap is executed, after which the software resets the value to 0 to prepare for the next input.

This process establishes a complete input handling pipeline, involving keyboard detection, USB protocol processing, driver-level register control, and hardware-level signal response. It enables real-time player interaction and effective keyboard-based control of in-game actions on the FPGA.

5 Future Work

In order to improve the immersion of the game, we plan to add an audio submodule in the future. The audio sounds required in the game are all encoded inside the audio generator, including: the collision sound when the bird hits the pipe; the sound of the bird flapping its wings; the sound of the game ending.

6 Contribution

Ethan Yang: USB keyboard, sprite rendering, vga controller

Tianshuo Jin: Pipe generation and scrolling, collision detection

Zidong Xu: Score module, background scrolling module, and part of images conversion.

Sijun Li: Image conversion, Sprite, Background scrolling module, Game State

7 System Verilog Code

```
module vga_ball(
    input logic        clk,
    input logic        reset,
    input logic [7:0]  writedata,
    input logic        write,
    input              chipselect,
    input logic [2:0]  address,

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

```
);

    // Game state definition
    typedef enum logic [1:0] {
        WAITING,   // Before first game starts
        PLAYING,   // Game is active
        GAME_OVER  // Bird has hit something
    } game_state_t;

    logic [10:0] hcount; // Horizontal pixel counter (0[U+FFFD]1023)
    logic [9:0]  vcount; // Vertical pixel counter (0[U+FFFD]511)

    logic [9:0] bird_y;  // Current vertical position of the bird
    logic [9:0] new_y;  // Updated vertical position after movement
    logic [1:0] bird_frame;  // Animation frame index for bird sprite
    logic [23:0] animation_counter;  // Counter for controlling animation
    speed

    logic [18:0] bg_addr;
    logic [7:0]  bg_color;

    logic [11:0] bird_addr;
    logic [7:0]  bird_color;

    logic [9:0] scroll_offset;  // Horizontal scrolling offset for
    background
    logic [23:0] scroll_counter;  // Counter to control scrolling speed

    logic [7:0] bird_color_reg;
    logic collision;
         logic game_over;

    logic [15:0] score;
    game_state_t game_state;

    // Gameover parameters
    parameter GAMEOVER_WIDTH = 192;
    parameter GAMEOVER_HEIGHT = 42;
    parameter GAMEOVER_X = 320 - GAMEOVER_WIDTH/2;
    parameter GAMEOVER_Y = 240 - GAMEOVER_HEIGHT/2;

    logic [15:0] gameover_addr;
    logic [7:0]  gameover_color;

    // === Score Display Start ===
```

```
localparam DIGIT_WIDTH  = 16;
localparam DIGIT_HEIGHT = 32;
localparam SEG_THICK    = 4;
localparam SCORE_X0     = 10;
localparam SCORE_Y0     = 10;
localparam SCORE_X1     = SCORE_X0 + DIGIT_WIDTH + 4;
localparam SCORE_Y1     = SCORE_Y0;

// BCD digits
logic [3:0] digit0, digit1;
// seven-segment decode signals {A,B,C,D,E,F,G}
logic [6:0] seg0, seg1;
// Pixel output for seven-segment display
logic       score_pixel;

parameter BIRD_X = 100;
parameter BIRD_WIDTH = 34;
parameter BIRD_HEIGHT = 24;

parameter GRAVITY = 1;
parameter FLAP_STRENGTH = -7;
parameter TEST_INTERVAL = 50_000_000;

logic signed [9:0] bird_velocity;
logic       vsync_reg, flap_latched;
logic [31:0] test_counter;

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)
);

bg_rom    bg_rom_inst (.address(bg_addr), .clock(clk), .q(bg_color));
bird_rom0 bird0       (.address(bird_addr), .clock(clk), .q(bird_color0)
);
bird_rom1 bird1       (.address(bird_addr), .clock(clk), .q(bird_color1)
);
```

```
bird_rom2 bird2       (.address(bird_addr), .clock(clk), .q(bird_color2)
);

     gameover_rom gameover_inst (.address(gameover_addr), .clock(clk), .
q(gameover_color));


logic [7:0] bird_color0, bird_color1, bird_color2;

function automatic bit in_rect(
    input int x0, input int y0,
    input int W,  input int H,
    input logic [10:0] hc,
    input logic [ 9:0] vc
);
    in_rect = (hc >= x0 && hc < x0 + W &&
               vc >= y0 && vc < y0 + H);
endfunction

always_ff @(posedge clk) begin
    case (bird_frame)
        2'd0: bird_color_reg <= bird_color0;
        2'd1: bird_color_reg <= bird_color1;
        2'd2: bird_color_reg <= bird_color2;
        default: bird_color_reg <= bird_color0;
    endcase
end

always_comb begin
    logic [9:0] bg_col;
              bg_col = (hcount[10:1] + scroll_offset) % 640;
              bg_addr = vcount * 640 + bg_col;
              /*
    bg_col = (hcount[10:1] + scroll_offset);
    if(bg_col < 640)
        bg_addr = vcount * 640 + bg_col;
    else
        bg_addr = vcount * 640 + (bg_col - 640);
                  */

    if (hcount[10:1] >= BIRD_X && hcount[10:1] < BIRD_X + BIRD_WIDTH &&
        vcount >= bird_y && vcount < bird_y + BIRD_HEIGHT)
        bird_addr = (vcount - bird_y) * BIRD_WIDTH + (hcount[10:1] -
BIRD_X);
    else
```

```
        bird_addr = 0;

              if (in_rect(GAMEOVER_X, GAMEOVER_Y, GAMEOVER_WIDTH,
GAMEOVER_HEIGHT, hcount[10:1], vcount))
        gameover_addr = (vcount - GAMEOVER_Y) * GAMEOVER_WIDTH + (
hcount[10:1] - GAMEOVER_X);
    else
        gameover_addr = 0;
end

//Pipe structure parameters
parameter PIPE_WIDTH = 52;
parameter GAP_HEIGHT = 100;
parameter PIPE_COUNT = 3;
parameter PIPE_SPACING = 213;

typedef struct packed {
    logic [9:0] x;
    logic [8:0] gap_y;
} pipe_t;

pipe_t pipes[PIPE_COUNT];

integer i;
logic [9:0] max_pipe_x;

// === LFSR LFSR Random Number Generation ===
logic [7:0] lfsr;
logic       lfsr_enable;


always_ff @(posedge clk or posedge reset) begin
    if (reset)
        lfsr <= 8'h5A;
    else if (lfsr_enable)
        lfsr <= {lfsr[6:0], lfsr[7] ^ lfsr[5]};
end

always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        bird_y <= 240;
        bird_velocity <= 0;
        vsync_reg <= 1'b0;
        flap_latched <= 1'b0;
        test_counter <= 32'd0;
```

```
        bird_frame <= 0;
        animation_counter <= 0;
        scroll_offset <= 0;
        scroll_counter <= 0;
        score <= 16'd0;
        game_state <= WAITING;

        for (i = 0; i < PIPE_COUNT; i = i + 1) begin
            pipes[i].x <= 640 + i * PIPE_SPACING;
            pipes[i].gap_y <= 150 + i * 40;
        end
    end else begin
        // Vertical sync edge detection
        vsync_reg <= VGA_VS;

        // Handle keyboard input (flapping)
        if (chipselect && write && address == 3'h7) begin
            // Register flap command from processor
            flap_latched <= writedata[0];
        end else if (VGA_VS && !vsync_reg && flap_latched) begin
            // Clear flap signal after frame update
            flap_latched <= 0;
        end

        // Animation counter for bird wings always updates
                            if (game_state != GAME_OVER) begin
                                    animation_counter <=
animation_counter + 1;
                                    if (animation_counter == 24'
d5_000_000) begin
                                            animation_counter <= 0;
                                            bird_frame <= (bird_frame
== 2) ? 0 : bird_frame + 1;
                                    end
                            end

        // Background always scrolls
                            if (game_state != GAME_OVER) begin
                                    scroll_counter <= scroll_counter +
1;
                                    if (scroll_counter == 24'd500_000)
begin
                                            scroll_offset <= (
scroll_offset + 1) % 640;
```

```
                                        /*
                                         scroll_offset <=
scroll_offset + 1;
                                         if (scroll_offset >= 640)
begin
                                                 scroll_offset <=
0;
                                         end
                                         */
                                         scroll_counter <= 0;
                                end
                            end

        // Game state specific logic
        case (game_state)
            WAITING: begin
                // Bird stays in the middle
                bird_y <= 240;
                bird_velocity <= 0;

                // Reset score
                score <= 0;

                // Reset pipes (initialize offscreen)
                for (i = 0; i < PIPE_COUNT; i = i + 1) begin
                    pipes[i].x <= 440 + i * PIPE_SPACING;
                    pipes[i].gap_y <= 150 + i * 20;
                end

                // Start game on flap
                if (flap_latched) begin
                    game_state <= PLAYING;
                    bird_velocity <= FLAP_STRENGTH; // Initial upward
velocity
                end
            end

            PLAYING: begin
                // Bird physics - update on vsync
                if (VGA_VS && !vsync_reg) begin

                    // Flap or apply gravity
                    if (flap_latched) begin

bird_velocity <= FLAP_STRENGTH; // Upward velocity
```

```
                    end else begin
                        bird_velocity <= bird_velocity + GRAVITY;
                    end

                    // Update position
                    new_y = bird_y + bird_velocity;

                    // Boundary checks
                    if (new_y < 0) begin
                        bird_y <= 0;
                    end else if (new_y >= 440 - BIRD_HEIGHT) begin
                        bird_y <= 440 - BIRD_HEIGHT;
                        bird_velocity <= 0;
                        game_state <= GAME_OVER; // Hit ground
                    end else begin
                        bird_y <= new_y;
                    end

                    // Find rightmost pipe
                    max_pipe_x = 0;
                    for (i = 0; i < PIPE_COUNT; i = i + 1)
                        if (pipes[i].x > max_pipe_x)
                            max_pipe_x = pipes[i].x;

                    // Update score when passing pipes
                    for (i = 0; i < PIPE_COUNT; i = i + 1) begin
                        //if (pipes[i].x + PIPE_WIDTH == BIRD_X) begin
                            //score <= score + 1;
                        //end
                                                                    if
 (pipes[i].x + PIPE_WIDTH < BIRD_X && pipes[i].x + PIPE_WIDTH >= BIRD_X
- 2) begin

          score <= score + 1;

end
                    end

                    // Move pipes and recycle them
                    for (i = 0; i < PIPE_COUNT; i = i + 1) begin
                        pipes[i].x <= pipes[i].x - 2; // Faster pipe
movement

                        if (pipes[i].x <= 1) begin
                            pipes[i].x <= max_pipe_x + PIPE_SPACING;
```

```
                            // Update random number generator for gap
position
                            lfsr_enable <= 1;
                            pipes[i].gap_y <= 80 + (lfsr % 160); //
Random gap position
                        end
                    end
                end else begin
                    lfsr_enable <= 0;
                end

                // Check for collisions
                if (collision) begin
                    game_state <= GAME_OVER;
                end
            end

            GAME_OVER: begin
                // Game over - bird stops, pipes stop
                bird_velocity <= 0;

                                            bird_y <= bird_y;

                // Wait for flap to restart
                if (flap_latched) begin
                    game_state <= WAITING;

flap_latched <= 0;
                end
            end
        endcase
    end
end

logic [15:0] ground_addr;
logic [7:0]  ground_color;

base_rom ground_inst (.address(ground_addr), .clock(clk), .q(
ground_color));

always_comb begin
    if (vcount >= 440 && vcount < 480)
        ground_addr = (vcount - 440) * 640 + ((hcount[10:1] +
scroll_offset) % 640);
    else
```

```
        ground_addr = 0;
end

logic pipe_pixel;

// === BCD Conversion ===
always_comb begin
    digit0 = (score / 10) % 10;
    digit1 = score % 10;
end

// === Seven-Segment Decode ===
always_comb begin
    case (digit0)
        4'd0: seg0 = 7'b1111110;
        4'd1: seg0 = 7'b0110000;
        4'd2: seg0 = 7'b1101101;
        4'd3: seg0 = 7'b1111001;
        4'd4: seg0 = 7'b0110011;
        4'd5: seg0 = 7'b1011011;
        4'd6: seg0 = 7'b1011111;
        4'd7: seg0 = 7'b1110000;
        4'd8: seg0 = 7'b1111111;
        4'd9: seg0 = 7'b1111011;
        default: seg0 = 7'b0000000;
    endcase
    case (digit1)
        4'd0: seg1 = 7'b1111110;
        4'd1: seg1 = 7'b0110000;
        4'd2: seg1 = 7'b1101101;
        4'd3: seg1 = 7'b1111001;
        4'd4: seg1 = 7'b0110011;
        4'd5: seg1 = 7'b1011011;
        4'd6: seg1 = 7'b1011111;
        4'd7: seg1 = 7'b1110000;
        4'd8: seg1 = 7'b1111111;
        4'd9: seg1 = 7'b1111011;
        default: seg1 = 7'b0000000;
    endcase
end

// === Score Pixel Generation ===
always_comb begin
    score_pixel = 1'b0;
    // Digit0 segments
```

```
    if (seg0[6] && in_rect(SCORE_X0 + SEG_THICK, SCORE_Y0,
                           DIGIT_WIDTH - 2*SEG_THICK, SEG_THICK,
                           hcount[10:1], vcount))
        score_pixel = 1;
    // Other segment checks...
    // (keeping the existing segment rendering code)
    if (seg0[5] && in_rect(SCORE_X0 + DIGIT_WIDTH - SEG_THICK, SCORE_Y0
 + SEG_THICK,
                           SEG_THICK, DIGIT_HEIGHT/2 - SEG_THICK,
                           hcount[10:1], vcount))
        score_pixel = 1;
    if (seg0[4] && in_rect(SCORE_X0 + DIGIT_WIDTH - SEG_THICK, SCORE_Y0
 + DIGIT_HEIGHT/2,
                           SEG_THICK, DIGIT_HEIGHT/2 - SEG_THICK,
                           hcount[10:1], vcount))
        score_pixel = 1;
    if (seg0[3] && in_rect(SCORE_X0 + SEG_THICK, SCORE_Y0 +
DIGIT_HEIGHT - SEG_THICK,
                           DIGIT_WIDTH - 2*SEG_THICK, SEG_THICK,
                           hcount[10:1], vcount))
        score_pixel = 1;
    if (seg0[2] && in_rect(SCORE_X0, SCORE_Y0 + DIGIT_HEIGHT/2,
                           SEG_THICK, DIGIT_HEIGHT/2 - SEG_THICK,
                           hcount[10:1], vcount))
        score_pixel = 1;
    if (seg0[1] && in_rect(SCORE_X0, SCORE_Y0 + SEG_THICK,
                           SEG_THICK, DIGIT_HEIGHT/2 - SEG_THICK,
                           hcount[10:1], vcount))
        score_pixel = 1;
    if (seg0[0] && in_rect(SCORE_X0 + SEG_THICK, SCORE_Y0 +
DIGIT_HEIGHT/2 - SEG_THICK/2,
                           DIGIT_WIDTH - 2*SEG_THICK, SEG_THICK,
                           hcount[10:1], vcount))
        score_pixel = 1;

    // Digit1 segments
    // (keeping the existing digit1 segment rendering code)
    if (seg1[6] && in_rect(SCORE_X1 + SEG_THICK, SCORE_Y1,
                           DIGIT_WIDTH - 2*SEG_THICK, SEG_THICK,
                           hcount[10:1], vcount))
        score_pixel = 1;
    if (seg1[5] && in_rect(SCORE_X1 + DIGIT_WIDTH - SEG_THICK, SCORE_Y1
 + SEG_THICK,
                           SEG_THICK, DIGIT_HEIGHT/2 - SEG_THICK,
                           hcount[10:1], vcount))
```

```
            score_pixel = 1;
        if (seg1[4] && in_rect(SCORE_X1 + DIGIT_WIDTH - SEG_THICK, SCORE_Y1
 + DIGIT_HEIGHT/2,
                                SEG_THICK, DIGIT_HEIGHT/2 - SEG_THICK,
                                hcount[10:1], vcount))
            score_pixel = 1;
        if (seg1[3] && in_rect(SCORE_X1 + SEG_THICK, SCORE_Y1 +
DIGIT_HEIGHT - SEG_THICK,
                                DIGIT_WIDTH - 2*SEG_THICK, SEG_THICK,
                                hcount[10:1], vcount))
            score_pixel = 1;
        if (seg1[2] && in_rect(SCORE_X1, SCORE_Y1 + DIGIT_HEIGHT/2,
                                SEG_THICK, DIGIT_HEIGHT/2 - SEG_THICK,
                                hcount[10:1], vcount))
            score_pixel = 1;
        if (seg1[1] && in_rect(SCORE_X1, SCORE_Y1 + SEG_THICK,
                                SEG_THICK, DIGIT_HEIGHT/2 - SEG_THICK,
                                hcount[10:1], vcount))
            score_pixel = 1;
        if (seg1[0] && in_rect(SCORE_X1 + SEG_THICK, SCORE_Y1 +
DIGIT_HEIGHT/2 - SEG_THICK/2,
                                DIGIT_WIDTH - 2*SEG_THICK, SEG_THICK,
                                hcount[10:1], vcount))
            score_pixel = 1;
    end

    // Pipe pixel detection
    always_comb begin
        pipe_pixel = 0;
        // Only show pipes when not in WAITING state
        if (game_state != WAITING) begin
            for (int j = 0; j < PIPE_COUNT; j = j + 1) begin
                if (hcount[10:1] >= pipes[j].x && hcount[10:1] < pipes[j].x
 + PIPE_WIDTH) begin
                    if ((vcount < pipes[j].gap_y || vcount > pipes[j].gap_y
 + GAP_HEIGHT) &&
                        vcount < 440)
                        pipe_pixel = 1;
                end
            end
        end
    end

    // Collision detection logic
    always_comb begin
```

```
    collision = 0;

    // Only check collisions in PLAYING state
    if (game_state == PLAYING) begin
        // Upper boundary collision
        //if (bird_y < 0)
        //    collision = 1;

        // Ground collision
        if (bird_y + BIRD_HEIGHT > 440)
            collision = 1;

        // Pipe collisions
        for (int j = 0; j < PIPE_COUNT; j = j + 1) begin
            // Check X overlap
            if (BIRD_X + BIRD_WIDTH > pipes[j].x &&
                BIRD_X < pipes[j].x + PIPE_WIDTH) begin

                // Check Y overlap (not in gap)
                if ((bird_y < pipes[j].gap_y && bird_y + BIRD_HEIGHT >
0) ||
                    bird_y + BIRD_HEIGHT > pipes[j].gap_y + GAP_HEIGHT)
                    collision = 1;
            end
        end
    end
end

// Rendering logic
always_comb begin
    {VGA_R, VGA_G, VGA_B} = 24'h000000;

    if (VGA_BLANK_n) begin
        // Render score (highest priority)
                            if (game_state == GAME_OVER && in_rect(
GAMEOVER_X, GAMEOVER_Y, GAMEOVER_WIDTH, GAMEOVER_HEIGHT, hcount[10:1],
vcount) &&
            gameover_color != 8'h00) begin
            VGA_R = {gameover_color[7:5], 5'b00000};
            VGA_G = {gameover_color[4:2], 5'b00000};
            VGA_B = {gameover_color[1:0], 6'b000000};
                            //rendering score
        end else if (score_pixel) begin
            VGA_R = 8'hFF;
            VGA_G = 8'hFF;
```

```
                VGA_B = 8'hFF;
            end
            // Render bird
            else if (hcount[10:1] >= BIRD_X && hcount[10:1] < BIRD_X +
    BIRD_WIDTH &&
                     vcount >= bird_y && vcount < bird_y + BIRD_HEIGHT &&
                     bird_color_reg != 8'h00) begin
                VGA_R = {bird_color_reg[7:5], 5'b00000};
                VGA_G = {bird_color_reg[4:2], 5'b00000};
                VGA_B = {bird_color_reg[1:0], 6'b000000};
            end
            // Render pipes (only if not in WAITING state)
            else if (pipe_pixel) begin
                VGA_R = 8'h00;
                VGA_G = 8'hFF;
                VGA_B = 8'h00;
            end
            // Render ground
            else if (vcount >= 440 && vcount < 480) begin
                VGA_R = {ground_color[7:5], 5'b00000};
                                        VGA_G = {ground_color[4:2], 5'
    b00000};
                                        VGA_B = {ground_color[1:0], 6'
    b000000};
            end
            // Render background
            else begin
                VGA_B = {bg_color[7:5], 5'b00000};
                VGA_G = {bg_color[4:2], 5'b00000};
                VGA_R = {bg_color[1:0], 6'b000000};
            end

                                /*
            // Add a red tint in GAME_OVER state
            if (game_state == GAME_OVER) begin
                VGA_R = VGA_R | 8'h40; // Add some red tint
            end
                                */
        end
    end
endmodule

module vga_counters(
 input logic         clk50, reset,
 output logic [10:0] hcount,  // hcount[10:1] is pixel column
```

```
 output logic [9:0]  vcount,  // vcount[9:0] is pixel row
 output logic        VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
 *
 * HCOUNT 1599 0             1279       1599 0
 *             ---------------              --------
 * ____________|    Video      |____________|  Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *       -----------------------      -------------
 * |____|       VGA_HS          |____|
 */
   // Parameters for hcount
   parameter HACTIVE      = 11'd 1280,
             HFRONT_PORCH = 11'd 32,
             HSYNC        = 11'd 192,
             HBACK_PORCH  = 11'd 96,
             HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC +
                            HBACK_PORCH; // 1600

   // Parameters for vcount
   parameter VACTIVE      = 10'd 480,
             VFRONT_PORCH = 10'd 10,
             VSYNC        = 10'd 2,
             VBACK_PORCH  = 10'd 33,
             VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC +
                            VBACK_PORCH; // 525

   logic endOfLine;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)          hcount <= 0;
     else if (endOfLine) hcount <= 0;
     else                hcount <= hcount + 11'd 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 + 10'd 1;

    assign endOfField = vcount == VTOTAL - 1;

    // Horizontal sync: from 0x520 to 0x5DF (0x57F)
    // 101 0010 0000 to 101 1101 1111
    assign VGA_HS = !( (hcount[10:8] == 3'b101) &
                       !(hcount[7:5] == 3'b111));
    assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

    assign VGA_SYNC_n = 1'b0; // For putting sync on the green signal;
     unused

    // Horizontal active: 0 to 1279     Vertical active: 0 to 479
    // 101 0000 0000  1280              01 1110 0000  480
    // 110 0011 1111  1599              10 0000 1100  524
    assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
                         !( vcount[9] | (vcount[8:5] == 4'b1111) );

    /* VGA_CLK is 25 MHz
     *             __    __    __
     * clk50    __|  |__|  |__|
     *
     *             _____       __
     * hcount[0]__|     |_____|
     */
    assign VGA_CLK = hcount[0]; // 25 MHz clock: rising edge sensitive

endmodule
```

Listing 1: vga_ball.sv

8 C Code

hello.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include "usbkeyboard.h"
#include "vga_ball.h"
```

```c
#define FLAP_KEY 0x2C  // USB keycode for spacebar
#define ESC_KEY  0x29  // USB keycode for ESC
#define DEVICE_FILE "/dev/vga_ball"

int main() {
    struct libusb_device_handle *keyboard;
    struct usb_keyboard_packet packet;
    uint8_t endpoint_address;
    int transferred;
    int vga_fd;

    // Open USB keyboard
    printf("Opening USB keyboard...\n");
    keyboard = openkeyboard(&endpoint_address);
    if (!keyboard) {
        fprintf(stderr, "Could not find a USB keyboard.\n");
        return 1;
    }

    // Open memory-mapped peripheral
    printf("Opening VGA Ball device...\n");
    vga_fd = open(DEVICE_FILE, O_RDWR);
    if (vga_fd < 0) {
        perror("Failed to open /dev/vga_ball");
        libusb_close(keyboard);
        return 1;
    }

    printf("Press SPACE to flap the bird. Press ESC to quit.\n");

    // Main loop
    while (1) {
        // Poll keyboard with a timeout of 10ms
        int result = libusb_interrupt_transfer(keyboard, endpoint_address,
            (unsigned char *)&packet, sizeof(packet),
            &transferred, 10);

        if (result == 0 && transferred == sizeof(packet)) {
            uint8_t code = packet.keycode[0];

            if (code == FLAP_KEY) {
                printf("SPACEBAR detected! Sending flap command...\n");

                // Trigger the flap command to FPGA using IOCTL
                vga_ball_arg_t vla = {0};  // Initialize all fields to 0
```

```
                vla.flap = 1;               // Set flap bit

                if (ioctl(vga_fd, VGA_BALL_WRITE_FLAP, &vla) == -1) {
                    perror("ioctl(VGA_BALL_WRITE_FLAP) failed");
                } else {
                    printf("Flap triggered successfully!\n");
                                        usleep(20000);  // 20ms should be
    enough for at least one frame

                                        // Reset flap signal
                                        vla.flap = 0;
                                        ioctl(vga_fd, VGA_BALL_WRITE_FLAP,
    &vla);
                }
            }

            if (code == ESC_KEY) {
                printf("Exiting...\n");
                break;
            }
        } else if (result != 0 && result != LIBUSB_ERROR_TIMEOUT) {
            // Only report non-timeout errors
            fprintf(stderr, "libusb_interrupt_transfer error: %d\n", result)
    ;
        }

        usleep(10000);  // 10ms sleep to avoid CPU hogging
    }

    close(vga_fd);
    libusb_close(keyboard);
    return 0;
}
```

vga.c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
```

```
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "vga_ball.h"

#define DRIVER_NAME "vga_ball"

#define BG_RED(x) (x)
#define BG_GREEN(x) ((x)+1)
#define BG_BLUE(x) ((x)+2)
#define BALL_X_LOW(x) ((x)+3)
#define BALL_X_HIGH(x) ((x)+4)
#define BALL_Y_LOW(x) ((x)+5)
#define BALL_Y_HIGH(x) ((x)+6)
#define FLAP_SIGNAL(x) ((x)+7)

struct vga_ball_dev {
        struct resource res;
        void __iomem *virtbase;
    vga_ball_color_t background;
        vga_ball_position_t ball;
        unsigned char flap;
} dev;

static void write_background(vga_ball_color_t *background) {
        iowrite8(background->red, BG_RED(dev.virtbase));
        iowrite8(background->green, BG_GREEN(dev.virtbase));
        iowrite8(background->blue, BG_BLUE(dev.virtbase));
        dev.background = *background;
}

static void write_ball_position(vga_ball_position_t *ball) {
        iowrite8(ball->x & 0xff, BALL_X_LOW(dev.virtbase));
        iowrite8((ball->x >> 8) & 0x03, BALL_X_HIGH(dev.virtbase));
        iowrite8(ball->y & 0xff, BALL_Y_LOW(dev.virtbase));
        iowrite8((ball->y >> 8) & 0x03, BALL_Y_HIGH(dev.virtbase));
        dev.ball = *ball;
}

static void write_flap(unsigned char flap) {
        unsigned char value = flap ? 1 : 0;
        iowrite8(value, FLAP_SIGNAL(dev.virtbase));
        dev.flap = value;
```

```
        printk(KERN_INFO "vga_ball: Wrote flap value %d to register 7\n",
    value);
}

static long vga_ball_ioctl(struct file *f, unsigned int cmd, unsigned long
    arg) {
        vga_ball_arg_t vla;

        switch (cmd) {
        case VGA_BALL_WRITE_BACKGROUND:
                if (copy_from_user(&vla, (vga_ball_arg_t *) arg, sizeof(
    vga_ball_arg_t)))
                        return -EACCES;
                write_background(&vla.background);
                break;

        case VGA_BALL_READ_BACKGROUND:
                vla.background = dev.background;
                if (copy_to_user((vga_ball_arg_t *) arg, &vla, sizeof(
    vga_ball_arg_t)))
                        return -EACCES;
                break;

        case VGA_BALL_WRITE_BALL:
                if (copy_from_user(&vla, (vga_ball_arg_t *) arg, sizeof(
    vga_ball_arg_t)))
                        return -EACCES;
                write_ball_position(&vla.ball);
                break;

        case VGA_BALL_READ_BALL:
                vla.ball = dev.ball;
                if (copy_to_user((vga_ball_arg_t *) arg, &vla, sizeof(
    vga_ball_arg_t)))
                        return -EACCES;
                break;

        case VGA_BALL_WRITE_FLAP:
                if (copy_from_user(&vla, (vga_ball_arg_t *) arg, sizeof(
    vga_ball_arg_t)))
                        return -EACCES;
                write_flap(vla.flap);
                break;

        default:
```

```
                return -EINVAL;
        }

        return 0;
}

static const struct file_operations vga_ball_fops = {
        .owner = THIS_MODULE,
        .unlocked_ioctl = vga_ball_ioctl,
};

static struct miscdevice vga_ball_misc_device = {
        .minor = MISC_DYNAMIC_MINOR,
        .name = DRIVER_NAME,
        .fops = &vga_ball_fops,
};

static int __init vga_ball_probe(struct platform_device *pdev) {
        vga_ball_color_t beige = { 0xf9, 0xe4, 0xb7 };
        vga_ball_position_t initial_ball = { 320, 240, 20 };
        int ret;

        ret = misc_register(&vga_ball_misc_device);
        ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
        if (ret) goto out_deregister;

        if (request_mem_region(dev.res.start, resource_size(&dev.res),
    DRIVER_NAME) == NULL) {
                ret = -EBUSY;
                goto out_deregister;
        }

        dev.virtbase = of_iomap(pdev->dev.of_node, 0);
        if (dev.virtbase == NULL) {
                ret = -ENOMEM;
                goto out_release_mem_region;
        }

        write_background(&beige);
        write_ball_position(&initial_ball);
        dev.flap = 0;
        write_flap(0);
        return 0;

out_release_mem_region:
```

```
        release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
        misc_deregister(&vga_ball_misc_device);
        return ret;
}

static int vga_ball_remove(struct platform_device *pdev) {
        iounmap(dev.virtbase);
        release_mem_region(dev.res.start, resource_size(&dev.res));
        misc_deregister(&vga_ball_misc_device);
        return 0;
}

#ifdef CONFIG_OF
static const struct of_device_id vga_ball_of_match[] = {
        { .compatible = "csee4840,vga_ball-1.0" },
        {},
};
MODULE_DEVICE_TABLE(of, vga_ball_of_match);
#endif

static struct platform_driver vga_ball_driver = {
        .driver = {
                .name = DRIVER_NAME,
                .owner = THIS_MODULE,
                .of_match_table = of_match_ptr(vga_ball_of_match),
        },
        .remove = __exit_p(vga_ball_remove),
};

static int __init vga_ball_init(void) {
        pr_info(DRIVER_NAME ": init\n");
        return platform_driver_probe(&vga_ball_driver, vga_ball_probe);
}

static void __exit vga_ball_exit(void) {
        platform_driver_unregister(&vga_ball_driver);
        pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_ball_init);
module_exit(vga_ball_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
```

```
MODULE_DESCRIPTION("VGA ball driver");
```

vga.h

```
#ifndef _VGA_BALL_H
#define _VGA_BALL_H
#include <linux/ioctl.h>

typedef struct {
    unsigned char red, green, blue;
} vga_ball_color_t;

typedef struct {
    unsigned short x, y;
    unsigned char radius;
} vga_ball_position_t;

typedef struct {
    vga_ball_color_t background;
    vga_ball_position_t ball;
    unsigned char flap;  // for passing flap input
} vga_ball_arg_t;

#define VGA_BALL_MAGIC 'q'
/* ioctls and their arguments */
#define VGA_BALL_WRITE_BACKGROUND _IOW(VGA_BALL_MAGIC, 1, vga_ball_arg_t)
#define VGA_BALL_READ_BACKGROUND _IOR(VGA_BALL_MAGIC, 2, vga_ball_arg_t)
#define VGA_BALL_WRITE_BALL _IOW(VGA_BALL_MAGIC, 3, vga_ball_arg_t)
#define VGA_BALL_READ_BALL _IOR(VGA_BALL_MAGIC, 4, vga_ball_arg_t)
#define VGA_BALL_WRITE_FLAP _IOW(VGA_BALL_MAGIC, 5, vga_ball_arg_t)

#endif
```

usbkeyboard.c

```
#include "usbkeyboard.h"
#include <stdio.h>
#include <stdlib.h>

/* References on libusb 1.0 and the USB HID/keyboard protocol
 * http://libusb.org
 * https://web.archive.org/web/20210302095553/https://www.dreamincode.net/
    forums/topic/148707-introduction-to-using-libusb-10/
 * https://www.usb.org/sites/default/files/documents/hid1_11.pdf
 * https://usb.org/sites/default/files/hut1_5.pdf
```

```
 */

/*
 * Find and return a USB keyboard device or NULL if not found
 */
struct libusb_device_handle *openkeyboard(uint8_t *endpoint_address) {
  libusb_device **devs;
  struct libusb_device_handle *keyboard = NULL;
  struct libusb_device_descriptor desc;
  ssize_t num_devs, d;
  uint8_t i, k;

  if ( libusb_init(NULL) < 0 ) {
    fprintf(stderr, "Error: libusb_init failed\n");
    exit(1);
  }

  if ( (num_devs = libusb_get_device_list(NULL, &devs)) < 0 ) {
    fprintf(stderr, "Error: libusb_get_device_list failed\n");
    exit(1);
  }

  for (d = 0 ; d < num_devs ; d++) {
    libusb_device *dev = devs[d];
    if ( libusb_get_device_descriptor(dev, &desc) < 0 ) {
      fprintf(stderr, "Error: libusb_get_device_descriptor failed\n");
      exit(1);
    }

    if (desc.bDeviceClass == LIBUSB_CLASS_PER_INTERFACE) {
      struct libusb_config_descriptor *config;
      libusb_get_config_descriptor(dev, 0, &config);
      for (i = 0 ; i < config->bNumInterfaces ; i++)
      for (k = 0 ; k < config->interface[i].num_altsetting ; k++) {
        const struct libusb_interface_descriptor *inter =
          config->interface[i].altsetting + k;
        if ( inter->bInterfaceClass == LIBUSB_CLASS_HID &&
             inter->bInterfaceProtocol == USB_HID_KEYBOARD_PROTOCOL) {
          int r;
          if ((r = libusb_open(dev, &keyboard)) != 0) {
            fprintf(stderr, "Error: libusb_open failed: %d\n", r);
            exit(1);
          }
          if (libusb_kernel_driver_active(keyboard,i))
            libusb_detach_kernel_driver(keyboard, i);
```

```
          libusb_set_auto_detach_kernel_driver(keyboard, i);
          if ((r = libusb_claim_interface(keyboard, i)) != 0) {
            fprintf(stderr, "Error: libusb_claim_interface failed: %d\n", r)
    ;
            exit(1);
          }
          *endpoint_address = inter->endpoint[0].bEndpointAddress;
          goto found;
        }
      }
    }
  }

found:
  libusb_free_device_list(devs, 1);
  return keyboard;
}
```

usbkeyboard.h

```
#ifndef _USBKEYBOARD_H
#define _USBKEYBOARD_H

#include <libusb-1.0/libusb.h>

#define USB_HID_KEYBOARD_PROTOCOL 1

/* Modifier bits */
#define USB_LCTRL  (1 << 0)
#define USB_LSHIFT (1 << 1)
#define USB_LALT   (1 << 2)
#define USB_LGUI   (1 << 3)
#define USB_RCTRL  (1 << 4)
#define USB_RSHIFT (1 << 5)
#define USB_RALT   (1 << 6)
#define USB_RGUI   (1 << 7)

struct usb_keyboard_packet {
  uint8_t modifiers;
  uint8_t reserved;
  uint8_t keycode[6];
};

/* Find and open a USB keyboard device.  Argument should point to
   space to store an endpoint address.  Returns NULL if no keyboard
```

```
   device was found. */
extern struct libusb_device_handle *openkeyboard(uint8_t *);

#endif
```