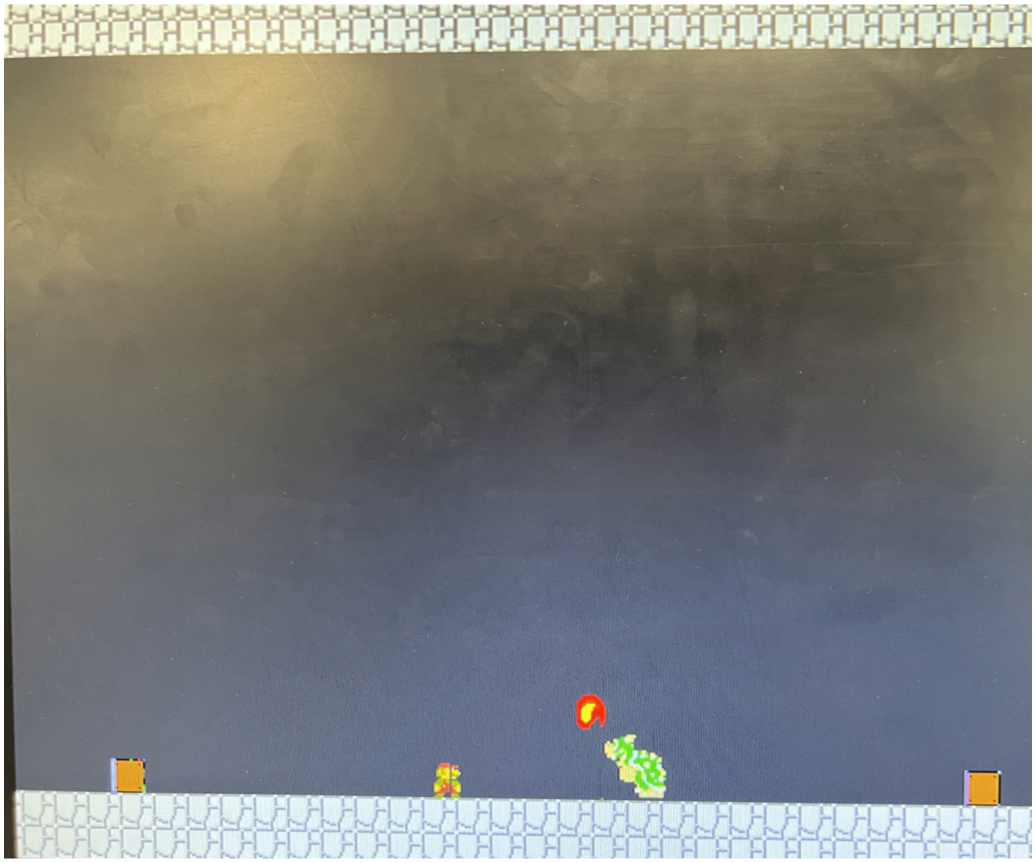
8-bit Mario Bowser Fight
Spring 2024

Prepared by
Brandon A. Khadan (bk2746), Bo Kizildag (bk2838), and Nico de la Cruz (nvd2109)

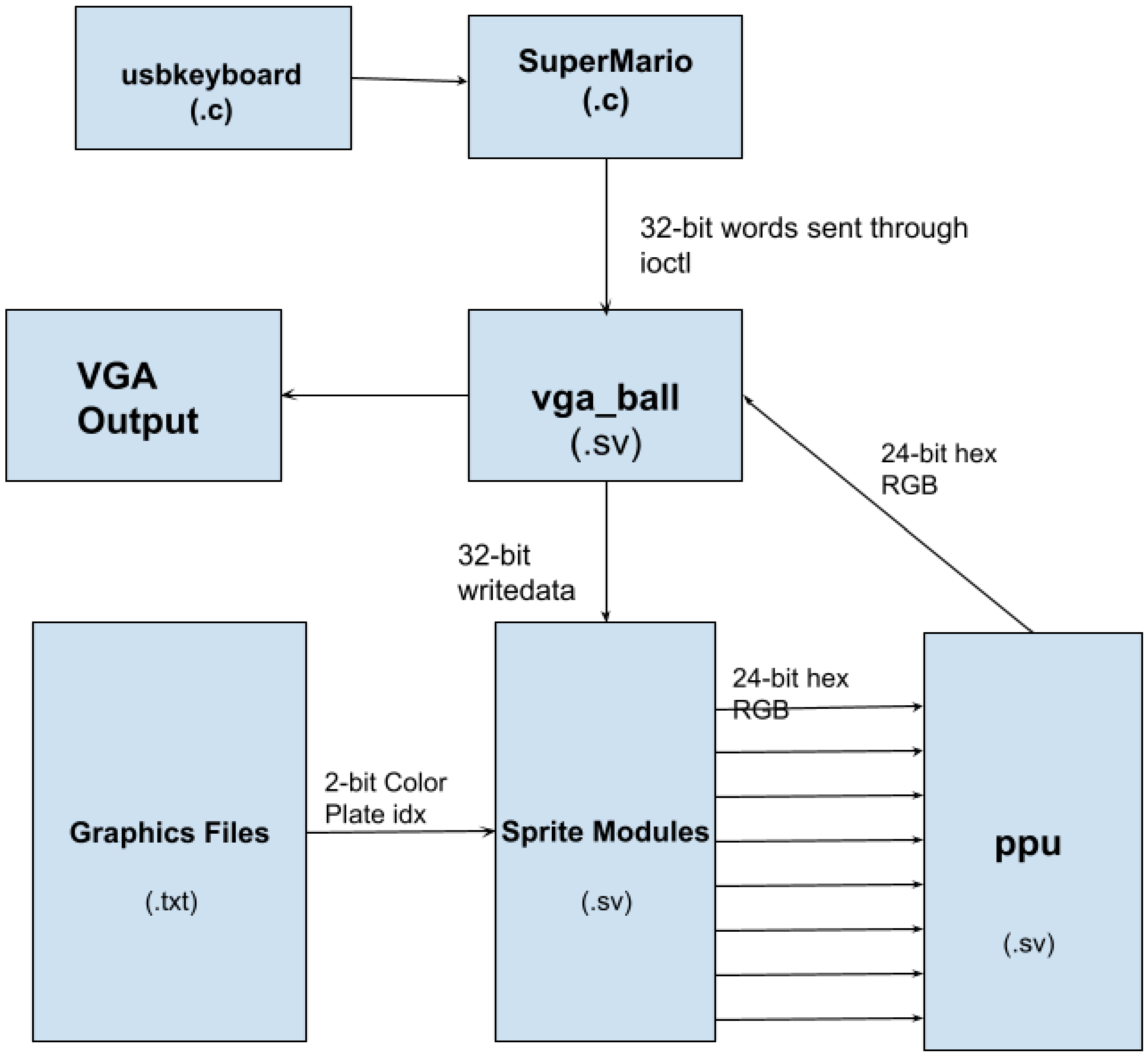
Table of Contents

Introduction

Super Mario Bros on NES is an iconic piece of gaming history. One memorable part of this game is the final boss fight against Bowser. In the original game, the player must navigate Mario around Bowser to reach a lever which can send Bowser down into a moat of lava. We decided to make our own interpretation of this boss fight where instead of simply running past Bowser, the player must actually engage him in combat. In our new and improved version, Bowser now has 3 hit points. Mario must run down this health bar before he can save the Princess. Our game is implemented on a Terasic DE1-SoC. Using System Verilog we created a hardware device that takes software configurable input to render the game out through VGA. Using C we implement a modular approach to controlling the game state and handle I/O to allow users a seamless experience.



System Diagram



Design

Software Design

The Entity Struct

```
typedef struct {
    PositionComponent position;
    MotionComponent motion;
    RenderComponent render;
    StateComponent state;
} Entity;
```

In designing this project, we sought a clean and concise way of representing controllable features in the game. To this end, we developed a structure called Entity that holds all the information the Software needs to manage the game state. Every attribute in the game including Mario, Ground/Ceiling, Bowser, etc. is represented as an Entity struct with 4 unique attributes: position, motion, render, and state.

```
typedef struct {
    float x, y;
    int width, height;
} PositionComponent;
```

The position struct holds the information needed to define where an object is in the game (x,y) and how large its interactable region or hitbox is (width, height).

```
typedef struct {
    float vx, vy;
    float ax, ay;
} MotionComponent;
```

The motion struct stores information on how the game should modify the position of the current Entity for the next frame. The important features are velocity (vx,vy) and acceleration (ax,ay) which enable fine tuned control of Entity motion in both dimensions of the screen.

```
typedef struct {
    uint32_t pattern_code;
    int visible;
    int flip;
} RenderComponent;
```

The render struct stores the information that will eventually be sent to the hardware. This includes the pattern_code which is an index to choose from a number of renderable sprite states (i.e. multiple kinds of blocks, large mario vs. small mario, etc). The mappings for each are stored in game_animation.h. The visible flag can be set to still perform software calculations but not render the entity. The flip flag is used to change the direction a sprite is facing. This is performed by changing the read order of the memory file.

```
typedef struct {
    int active;
    int state;
    int animate_frame_counter;
    int type;
} StateComponent;
```

The state component holds the software specific control information related to the Entity. The active flag tells the software whether the Entity should be calculated when rendering a frame. The state flag holds various state flags that define what type of Sprite this Entity is as defined in game_struct.h. The animate_frame_counter variable stores the current frame to allow for consistent timing between sprite states (ex. Mario walking animation consists of 3 different states spaced out in time).

Game Initialization

```
typedef struct {
    int camera_start;
    int camera_pos;
    float camera_velocity;
    int game_state;
    Entity entities[MAX_ENTITIES];
} Game;
```

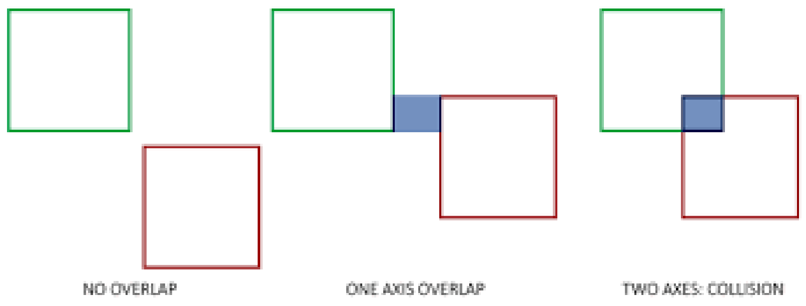
The game itself is stored in the Game struct. It holds the information needed to effectively implement the core features of entity management, camera scrolling and state management. All entities are stored in an array to allow for easy manipulation of Entities.

Game initialization is done by setting initial variables for the Game struct followed by creating and adding each Entity to the entities array.

```
void new_game(Game *game) {
    game->camera_pos = 70;
    game->camera_start = 70;
    game->game_state = GAME_START;
    game->camera_velocity = 0;

    // Initialize all entities to inactive
    for (int i = 0; i < MAX_ENTITIES; i++) {
        game->entities[i].state.active = 0;
    }

    // Initialize Mario
    Entity *mario = &game->entities[0];
    *mario = (Entity){
        .position = {128, 128, 16, 16},
        .motion = {0, 0, 0, 0},
        .render = {ANI_MARIO_S_NORMAL, 1, 0},
        .state = {1, STATE_NORMAL, 0, TYPE_MARIO_SMALL}
    };

    // Initialize Goomba
    Entity *goomba = &game->entities[1];
    *goomba = (Entity){
        .position = {300, GROUND_LEVEL - 16, 16, 16},
        .motion = {1.0, 0, 0, 0},
        .render = {ANI_GOOMBA_NORMAL, 1, 0},
        .state = {1, STATE_NORMAL, 0, TYPE_GOOMBA}
    };
```

Hitbox Detection

The hitbox detection algorithm determines if two entities are overlapping and identifies the direction of the least overlap between them. Each entity's hitbox is defined by its left, right, top, and bottom edges. The function first checks for non-overlapping cases; if any edge of one hitbox is beyond the corresponding edge of the other, no collision is detected. If an overlap exists, the function calculates the depth of overlap from each direction (left, right, top, bottom) between the two hitboxes. It then identifies which of these overlaps is minimal, indicating the shortest distance through which one hitbox can be moved out of collision with the other. Depending on this minimal overlap direction, the function returns a corresponding contact type enum, indicating the direction Entity A would need to move to resolve the collision with Entity B with the least amount of movement.



Camera Scrolling

Camera scrolling is done using the variables contained in the game struct. When mario moves across the median of the screen (defined in code as 66% of CAMERA_LIMIT) all entities will have a constant velocity applied to them to shift them to the left. Rightward movement is restricted to reduce the number of sprites that can be on the screen at once. This allows mario to move freely between the median line and the left side of the screen for a more interactive experience.

```
    mario->position.x += mario->motion.vx;
    game->camera_velocity = 0;
    if (mario->position.x > ((2*CAMERA_SIZE)/3)) {
        mario->position.x = ((2*CAMERA_SIZE)/3) - 1;
        game->camera_velocity = mario->motion.vx;

        if (game->camera_pos < 355 || bowser_alive == 0)
            game->camera_pos += mario->motion.vx;
        if (game->camera_pos < game->camera_start) {
            game->camera_pos = game->camera_start;
        }
    } else if (mario->position.x < 70) {
        mario->position.x = 70 + 1;
    }
```

```
                    default:
                        if (game.camera_pos < 355 || bowser_alive == 0) {
                            entity->position.x -= game.camera_velocity;
                        }
```

Hardware Communication

We chose 32-bit words to send information to the hardware through the ioctol.

```
void write_to_hardware(int vga_fd, int register_address, int data) {
    vga_ball_arg_t vla;
    vla.addr = register_address;
    vla.info = data;
    if (ioctl(vga_fd, VGA_BALL_WRITE_BACKGROUND, &vla) < 0) {
        fprintf(stderr, "Failed to write data to hardware\n");
    }
}
```

```
void flush_mario(const Entity *entity, int frame_select) {

    int visible = entity->render.visible;
    int flip = entity->render.flip;
    int x = entity->position.x;
    int y = entity->position.y;
    int pattern_code = entity->render.pattern_code;

    if (frame_counter % 100 == 0)
        printf("Flushing MARIO - Visible: %d, Flip: %d, X: %d, Y: %d, 
Pattern: %d\n", visible, flip, x, y, pattern_code);

    write_to_hardware(vga_ball_fd, 0, (int)((1 << 26) + (1 << 17) + 
(info_001 << 14) + (frame_select << 13) + (visible << 12) + (flip << 11) + 
(pattern_code & 0x1F)));
    write_to_hardware(vga_ball_fd, 0, (int)((1 << 26) + (1 << 17) + 
(info_010 << 14) + (frame_select << 13) + (x & 0x3FF)));
    write_to_hardware(vga_ball_fd, 0, (int)((1 << 26) + (1 << 17) + 
(info_011 << 14) + (frame_select << 13) + (y & 0x3FF)));
}
```

32-bit Word

Component Selector [31:26]: This segment of the writedata word is used to specify which component the data is intended for. This allows the routing of commands or data to specific parts of the display setup.

Child Component Selector [25:21]: When multiple instances of the same object need to be displayed, these bits can be used to address specific child components under a single main component that shares access to the memory file.

Action Code [20:17]: This field specifies the type of action to be performed. For example, resetting a sprite, changing its visibility, updating its position, or altering its appearance.

Action Type [16:14]: Further specifies the type of action, providing additional details or variations on the actions specified by the Action Code. For instance, different types of visibility changes or different animation frames could be selected using this field.

Buffer Toggle [13]: This bit can be used to toggle between different buffers (double buffering approach) or could indicate a special mode of operation.

Action Data [12:0]: The payload of the writedata. This part carries the actual data required for the action. This could be new coordinates for a sprite, new pattern indices, colors, or other parameters needed by the display logic.

User I/O

```
void *input_thread_function(void *ignored)
{
    struct usb_keyboard_packet packet;
    int transferred;
    int r;
    struct timeval timeout = { 0, 500000 };
    uint8_t first, second, chosen;

    for (;;) {
        r = libusb_interrupt_transfer(keyboard, endpoint_address,
(unsigned char *)&packet, sizeof(packet), &transferred, 0);
        if (r == 0 && transferred == sizeof(packet)) {

            first = packet.keycode[0];
            second = packet.keycode[1];
            chosen = 0;

            if (first != 0 && second != 0) {

                if (first == second) {
                    usleep(5000);
                    continue;
                } else {
                    chosen = second;
                }

            } else {
                chosen = first;
            }

            switch(chosen) {
                case 0x2C:
                    current_key = KEY_JUMP;
                    break;
                case 0x04:
                    current_key = KEY_LEFT;
                    break;
                case 0x07:
                    current_key = KEY_RIGHT;
                    break;
                case 0x0A:
                    current_key = KEY_NEWGAME;
                    break;
                default:
                    current_key = KEY_NONE;
                    break;
            }
        } else {
            if (r == LIBUSB_ERROR_NO_DEVICE) {

                fprintf(stderr, "Keyboard disconnected.\n");
                break;
            }

            fprintf(stderr, "Transfer error: %s\n", libusb_error_name(r));
            current_key = KEY_NONE;
            libusb_handle_events_timeout(NULL, &timeout);
        }
    }
    return NULL;
}
```

A keyboard is used to handle user input and output. Our version of the game uses the A, D, SPACE, and G keys to handle all user controls. The keyboard input is interpreted by the libusb library that sends out packets with ascii values representing the different key presses. These keys will set a global variable current_key using an enum value to denote different actions.

Hardware

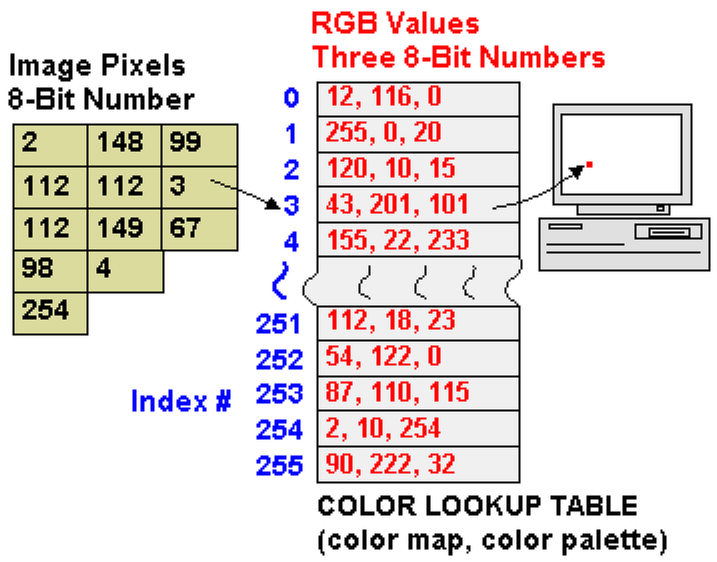
Pixel Processing Unit (ppu)

```
    );
    Ground_display Ground_0(
        .clk(clk), .reset(reset), .writedata(writedata),
        .hcount(hcount), .vcount(vcount), .RGB_output(RGB_list[6])
    );
    Tube_display Tube_0(
        .clk(clk), .reset(reset), .writedata(writedata),
        .hcount(hcount), .vcount(vcount), .RGB_output(RGB_list[7])
    );

    always_comb begin
        RGB_output = BACKGROUND_COLOR;
        for (int i = 0; i < 8; i = i + 1) begin
            if (RGB_list[i] != BACKGROUND_COLOR) begin
                RGB_output = RGB_list[i];
                break;
            end
        end
    end
```

The ppu consists of combinational logic that will determine whether a given pixel location (hcount, vcount) will render the background color or the output of one the sprites if one draws in that location. Naturally this logic models a hierarchical approach with Mario always taking first precedence as the first entry in RGB_output and Tube being the last since in the game it renders from under the ground.

Entity Module

```
    parameter [5:0] COMPONENT_ID = 6'b001001; // 9
    parameter [15:0] addr_limit = 16'd2048; // Limit for memory addressing
    logic [1:0] mem [0:2047]; // Memory for color indices
    logic [23:0] color_palette [0:4];
    logic [79:0] pattern_table [0:0];

    assign color_palette[0] = 24'hFFCC66; // Light Brown
    assign color_palette[1] = 24'h33CC33; // Green
    assign color_palette[2] = 24'hFFFFFF; // White
    assign color_palette[3] = 24'h202020; // Dark Gray
    assign color_palette[4] = 24'h000000; // Black


    assign pattern_table[0] = {16'd0, 16'd32, 16'd32, 16'd32, 16'd32}; //
Append, Res H, Res V, Act H, Act V
/* More code */
    always_comb begin
        if (buffer_valid[buffer_select] &&
buffer_address_output[buffer_select] < addr_limit) begin
            RGB_output =
color_palette[mem[buffer_address_output[buffer_select]]]; // Fetch color
using right-shift for 2-bit color index
        end else begin
            RGB_output = 24'h202020; // Default background color
        end
    end
    initial begin

$readmemh("/user/stud/fall21/bk2746/Projects/EmbeddedLab/Project_hw/on_chi
p_mem/Bowser.txt", mem);
    end
```



It is the job of each of the entity modules to control access to the memory file and output the correct color for the given location of the sprite. A double buffering approach is employed to to minimize visual artifacts such as flickering, tearing, and frame stuttering, which can occur when updating the screen. Each memory file referenced contains 2-bit values that correspond to indices in the color_palette.

Memory File Outline

File	Bytes
Mario.txt	10 KB
Bowser.txt	2 KB
Peach.txt	2.5 KB
Block.txt	4.5 KB
Tube.txt	0.9 KB
Ground.txt	0.382 KB
Goomba.txt	0.57 KB

Audio Attempt

The octave is divided into 12 logarithmically equal steps, each step being a semitone. This division means that the frequency ratio between any two adjacent notes (like C and C#, or E and F) is the twelfth root of two (21/1221/12), approximately 1.05946. This system allows for consistent intervals across keys, which is essential for the flexibility in modulation and transposition in modern music composition and performance.

Calculating frequencies

To calculate the frequencies of the other notes from the reference pitch A4 = 440 Hz, gotta use the formula: Frequency of Note=440×2(*n*12)Frequency of Note=440×2(12n) where *n*n is the number of semitones away from A4. If *n*n is positive, the note is higher than A4; if *n*n is negative, the note is lower.
Example:
C4 (Middle C) is 9 semitones below A4. Hence its frequency is:
440×2−(912)≈261.63 Hz440×2−(129)≈261.63 Hz
D4 is 7 semitones below A4, so: 440×2−(712)≈293.66 Hz
440×2−(127)≈293.66 Hz
E4 is 5 semitones below A4, so: 440×2−(512)≈329.63 Hz
440×2−(125)≈329.63 Hz
F4 is 4 semitones below A4, so: 440×2−(412)≈349.23 Hz
440×2−(124)≈349.23 Hz
G4 is 2 semitones below A4, so: 440×2−(212)≈392.00 Hz
440×2−(122)≈392.00 Hz

Tone generation is done via a counter to create a square wave at a specific frequency determined by the half_period input

The module utilizes an internal counter that increments on every clock cycle when note_enable is high

When the counter reaches the half_period value, the output (note_out) toggles, creating a square wave

The frequency of the square wave is determined by how quickly the counter reaches the half_period value, setting the tone's pitch

Contributions

- Brandon Khadan (bk2746): Software Design, Sprite Creation and PPU optimization, Game Logic, I/O Optimization
- Bo Kizildag (bk2838): Game Audio, .PNG processing, Design Documentation, Level Design
- Nico de la Cruz (nvd2109): Sprite collection, Documentation

Code

Hardware

Bowser Display:

```
module Bowser_display (
    input logic        clk,
    input logic        reset,
    input logic [31:0] writedata,
    input logic [9:0]  hcount,
    input logic [9:0]  vcount,
    output logic [23:0] RGB_output
);

    parameter [5:0] COMPONENT_ID = 6'b001001; // 9
    parameter [15:0] addr_limit = 16'd2048; // Limit for memory addressing
    logic [1:0] mem [0:2047]; // Memory for color indices
    logic [23:0] color_palette [0:4];
    logic [79:0] pattern_table [0:0];

    // Setup color palette
    assign color_palette[0] = 24'hFFCC66; // Light Brown
    assign color_palette[1] = 24'h33CC33; // Green
    assign color_palette[2] = 24'hFFFFFF; // White
    assign color_palette[3] = 24'h202020; // Dark Gray
    assign color_palette[4] = 24'h000000; // Black

    // Pattern definition
    assign pattern_table[0] = {16'd0, 16'd32, 16'd32, 16'd32, 16'd32}; //
Append, Res H, Res V, Act H, Act V

    // Buffers for double buffering
    logic [23:0] buffer_color_output[0:1];
    logic [15:0] buffer_address_output[0:1];
    logic buffer_valid[0:1];
    logic [111:0] buffer_state[0:1];
    logic buffer_select = 1'b0;
    // Decode writedata fields
    logic [5:0] component;
    logic [3:0] action;
    logic [2:0] action_type;
    logic [12:0] action_data;
    logic buffer_toggle;

    assign component = writedata[31:26];
    assign action = writedata[20:17];
    assign action_type = writedata[16:14];
    assign buffer_toggle = writedata[13];
    assign action_data = writedata[12:0];

    // Address calculators for each buffer
    addr_cal addr_cal_ping(
        .pattern_info(buffer_state[0][111:32]),
        .sprite_info(buffer_state[0][31:0]),
        .hcount(hcount),
        .vcount(vcount),
        .addr_output(buffer_address_output[0]),
        .valid(buffer_valid[0])
    );

    addr_cal addr_cal_pong(
        .pattern_info(buffer_state[1][111:32]),
        .sprite_info(buffer_state[1][31:0]),
        .hcount(hcount),
        .vcount(vcount),
        .addr_output(buffer_address_output[1]),
        .valid(buffer_valid[1])
    );

    // Process input messages to control sprite parameters
    always_ff @(posedge clk) begin
        if (reset) begin
            buffer_select <= 0;
            buffer_state[0] <= 0;
            buffer_state[1] <= 0;
        end else begin
            case (action)
```

```
                4'b1111: begin  // Reset and toggle buffer
                    buffer_select <= buffer_toggle;
                    buffer_state[~buffer_toggle] <= 1'b0; // Clear
inactive buffer
                end
                4'h0001: if (component == COMPONENT_ID) begin
                    // Update buffer state based on input type
                    case (action_type)
                        3'b001: begin  // Set visibility and pattern
                            buffer_state[buffer_toggle][31:30] <=
{action_data[12], action_data[11]};
                            if (action_data[4:0] == 0) // Check for valid
pattern index
                                buffer_state[buffer_toggle][111:32] <=
pattern_table[action_data[4:0]];;
                        end
                        3'b010: buffer_state[buffer_toggle][29:20] <=
action_data[9:0]; // X position
                        3'b011: buffer_state[buffer_toggle][19:10] <=
action_data[9:0]; // Y position
                        3'b100: buffer_state[buffer_toggle][9:0] <=
action_data[9:0]; // Additional attributes (if any)
                    endcase
                end
            endcase
        end
    end

    // Determine RGB output based on active buffer state and validity
    always_comb begin
        if (buffer_valid[buffer_select] &&
buffer_address_output[buffer_select] < addr_limit) begin
            RGB_output =
color_palette[mem[buffer_address_output[buffer_select]]]; // Fetch color
using right-shift for 2-bit color index
        end else begin
            RGB_output = 24'h202020; // Default background color
        end
    end
    // Initialize pixel data from memory
    initial begin

$readmemh("/user/stud/fall21/bk2746/Projects/EmbeddedLab/Project_hw/on_chi
p_mem/Bowser.txt", mem);
    end
endmodule
```

Block Display:

```
/*
 * Avalon memory-mapped peripheral that generates VGA
 *
 * Stephen A. Edwards
 * Columbia University
 */

module Block_display (input logic        clk,
        input logic          reset,
        input logic [31:0]   writedata,
        input logic [9:0]    hcount,
        input logic [9:0]    vcount,

        output logic [23:0] RGB_output);

    parameter [5:0] COMPONENT_ID = 6'b000111; // 7
    parameter [4:0] pattern_num = 5'd_17;
    parameter [15:0] addr_limit = 16'd_2304;
    parameter [4:0] child_limit = 5'd_9;
    logic [3:0] mem [0:2303];
    logic [23:0] color_plate [0:9];
    logic [79:0] pattern_table [0:16];

    assign pattern_table[0] = {16'd_0, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
    assign pattern_table[1] = {16'd_256, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
    assign pattern_table[2] = {16'd_512, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
```

```
    assign pattern_table[3] = {16'd_768, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
    assign pattern_table[4] = {16'd_1024, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
    assign pattern_table[5] = {16'd_1280, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
    assign pattern_table[6] = {16'd_1536, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
    assign pattern_table[7] = {16'd_1792, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
    assign pattern_table[8] = {16'd_2048, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
    assign pattern_table[9] = {16'd_1280, 16'd_16, 16'd_16, 16'd_32,
16'd_16};
    assign pattern_table[10] = {16'd_1280, 16'd_16, 16'd_16, 16'd_48,
16'd_16};
    assign pattern_table[11] = {16'd_1280, 16'd_16, 16'd_16, 16'd_128,
16'd_16};
    assign pattern_table[12] = {16'd_2048, 16'd_16, 16'd_16, 16'd_16,
16'd_32};
    assign pattern_table[13] = {16'd_2048, 16'd_16, 16'd_16, 16'd_16,
16'd_48};
    assign pattern_table[14] = {16'd_2048, 16'd_16, 16'd_16, 16'd_16,
16'd_64};
    assign pattern_table[15] = {16'd_2048, 16'd_16, 16'd_16, 16'd_64,
16'd_64};
    assign pattern_table[16] = {16'd_2048, 16'd_16, 16'd_16, 16'd_16,
16'd_64};

    assign color_plate[0] = 24'h202020;
    assign color_plate[1] = 24'h908fff;
    assign color_plate[2] = 24'h9a4b00;
    assign color_plate[3] = 24'he69a25;
    assign color_plate[4] = 24'h000000;
    assign color_plate[5] = 24'h974f00;
    assign color_plate[6] = 24'h572200;
    assign color_plate[7] = 24'h1a9300;
    assign color_plate[8] = 24'hfce1ce;
    assign color_plate[9] = 24'hffcdc4;
    logic [23:0] buffer_RGB_output[0:1][0:8];
    logic [15:0] buffer_addr_output[0:1][0:8];
    logic        buffer_addr_out_valid[0:1][0:8];
    logic [111:0] buffer_stateholder[0:1][0:8];
    logic        buffer_select = 1'b0;

    logic [5:0] sub_comp;
    logic [4:0] child_comp;
    logic [3:0] info;
    logic [2:0] input_type;
    logic [12:0] input_msg;
    logic       buffer_select_signal;

    assign sub_comp = writedata[31:26];
    assign child_comp = writedata[25:21];
    assign info = writedata[20:17];
    assign input_type = writedata[16:14];
    assign buffer_select_signal = writedata[13];
    assign input_msg = writedata[12:0];

    integer i, j, k;

    addr_cal
AC_left_buffer_0(.pattern_info(buffer_stateholder[0][0][111:32]),
.sprite_info(buffer_stateholder[0][0][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[0][0]),
.valid(buffer_addr_out_valid[0][0]));
    addr_cal
AC_left_buffer_1(.pattern_info(buffer_stateholder[0][1][111:32]),
.sprite_info(buffer_stateholder[0][1][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[0][1]),
.valid(buffer_addr_out_valid[0][1]));
    addr_cal
AC_left_buffer_2(.pattern_info(buffer_stateholder[0][2][111:32]),
.sprite_info(buffer_stateholder[0][2][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[0][2]),
.valid(buffer_addr_out_valid[0][2]));
    addr_cal
AC_left_buffer_3(.pattern_info(buffer_stateholder[0][3][111:32]),
.sprite_info(buffer_stateholder[0][3][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[0][3]),
.valid(buffer_addr_out_valid[0][3]));
    addr_cal
AC_left_buffer_4(.pattern_info(buffer_stateholder[0][4][111:32]),
.sprite_info(buffer_stateholder[0][4][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[0][4]),
.valid(buffer_addr_out_valid[0][4]));
    addr_cal
AC_left_buffer_5(.pattern_info(buffer_stateholder[0][5][111:32]),
.sprite_info(buffer_stateholder[0][5][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[0][5]),
.valid(buffer_addr_out_valid[0][5]));
    addr_cal
AC_left_buffer_6(.pattern_info(buffer_stateholder[0][6][111:32]),
.sprite_info(buffer_stateholder[0][6][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[0][6]),
.valid(buffer_addr_out_valid[0][6]));
    addr_cal
AC_left_buffer_7(.pattern_info(buffer_stateholder[0][7][111:32]),
.sprite_info(buffer_stateholder[0][7][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[0][7]),
.valid(buffer_addr_out_valid[0][7]));
    addr_cal
AC_left_buffer_8(.pattern_info(buffer_stateholder[0][8][111:32]),
.sprite_info(buffer_stateholder[0][8][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[0][8]),
.valid(buffer_addr_out_valid[0][8]));

    addr_cal
AC_right_buffer_0(.pattern_info(buffer_stateholder[1][0][111:32]),
.sprite_info(buffer_stateholder[1][0][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[1][0]),
.valid(buffer_addr_out_valid[1][0]));
    addr_cal
AC_right_buffer_1(.pattern_info(buffer_stateholder[1][1][111:32]),
.sprite_info(buffer_stateholder[1][1][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[1][1]),
.valid(buffer_addr_out_valid[1][1]));
    addr_cal
AC_right_buffer_2(.pattern_info(buffer_stateholder[1][2][111:32]),
.sprite_info(buffer_stateholder[1][2][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[1][2]),
.valid(buffer_addr_out_valid[1][2]));
    addr_cal
AC_right_buffer_3(.pattern_info(buffer_stateholder[1][3][111:32]),
.sprite_info(buffer_stateholder[1][3][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[1][3]),
.valid(buffer_addr_out_valid[1][3]));
    addr_cal
AC_right_buffer_4(.pattern_info(buffer_stateholder[1][4][111:32]),
.sprite_info(buffer_stateholder[1][4][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[1][4]),
.valid(buffer_addr_out_valid[1][4]));
    addr_cal
AC_right_buffer_5(.pattern_info(buffer_stateholder[1][5][111:32]),
.sprite_info(buffer_stateholder[1][5][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[1][5]),
.valid(buffer_addr_out_valid[1][5]));
    addr_cal
AC_right_buffer_6(.pattern_info(buffer_stateholder[1][6][111:32]),
.sprite_info(buffer_stateholder[1][6][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[1][6]),
.valid(buffer_addr_out_valid[1][6]));
    addr_cal
AC_right_buffer_7(.pattern_info(buffer_stateholder[1][7][111:32]),
.sprite_info(buffer_stateholder[1][7][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[1][7]),
.valid(buffer_addr_out_valid[1][7]));
    addr_cal
AC_right_buffer_8(.pattern_info(buffer_stateholder[1][8][111:32]),
.sprite_info(buffer_stateholder[1][8][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[1][8]),
.valid(buffer_addr_out_valid[1][8]));

    always_ff @(posedge clk) begin
        case (info)

            4'b1111: begin
                buffer_select = buffer_select_signal;
                for (i = 0; i < child_limit; i = i + 1) begin
```

```
                    buffer_stateholder[~buffer_select_signal][i][31] =
1'b0;
                end

            end

            4'h0001 : begin

                if (sub_comp == COMPONENT_ID) begin
                    if (child_comp < child_limit) begin
                        case (input_type)
                            3'b001: begin
                                // visible

buffer_stateholder[buffer_select_signal][child_comp][31] = input_msg[12];
                                // fliped

buffer_stateholder[buffer_select_signal][child_comp][30] = input_msg[11];
                                // pattern code
                                if (input_msg[4:0] < pattern_num)

buffer_stateholder[buffer_select_signal][child_comp][111:32] =
pattern_table[input_msg[4:0]];
                            end
                            3'b010: begin
                                // x_coordinate

buffer_stateholder[buffer_select_signal][child_comp][29:20] =
input_msg[9:0];
                            end
                            3'b011: begin
                                // y_coordinate

buffer_stateholder[buffer_select_signal][child_comp][19:10] =
input_msg[9:0];
                            end
                            3'b100: begin
                                // shift_amount
```

```
buffer_stateholder[buffer_select_signal][child_comp][9:0] =
input_msg[9:0];
                            end
                        endcase
                    end
                end
            end
       endcase
    end

    always_comb begin
        for (j = 0; j < child_limit; j = j + 1) begin
            buffer_RGB_output[0][j] =  (buffer_addr_output[0][j] <
addr_limit)? color_plate[mem[buffer_addr_output[0][j]]] :
color_plate[mem[0]];

            buffer_RGB_output[1][j] =  (buffer_addr_output[1][j] <
addr_limit)? color_plate[mem[buffer_addr_output[1][j]]] :
color_plate[mem[0]];
        end

        RGB_output = 24'h202020;
        for (k = 0; k < child_limit; k = k + 1) begin
            if ((buffer_RGB_output[buffer_select][k] != 24'h202020) &&
buffer_addr_out_valid[buffer_select][k]) begin
                RGB_output = buffer_RGB_output[buffer_select][k];
                break;
            end
        end
    end

initial begin

$readmemh("/user/stud/fall21/bk2746/Projects/EmbeddedLab/Project_hw/on_chi
p_mem/Block.txt", mem);
end


endmodule
```

Fireball Display:

```
module Fireball_display (
    input logic        clk,
    input logic        reset,
    input logic [31:0] writedata,
    input logic [9:0]  hcount,
    input logic [9:0]  vcount,
    output logic [23:0] RGB_output
);

    parameter [5:0] COMPONENT_ID = 6'b001000; // 8
    parameter [15:0] addr_limit = 16'd512;
    logic [3:0] mem [0:511]; // Memory for color indices
    logic [23:0] color_palette [0:4];
    logic [79:0] pattern_table [0:0]; // Only one pattern in use

    // Setup color palette
    assign color_palette[0] = 24'hFF0000;
    assign color_palette[1] = 24'hFF3300;
    assign color_palette[2] = 24'hFFFF00;
    assign color_palette[3] = 24'h202020;

    // Pattern definition
    assign pattern_table[0] = {16'd0, 16'd14, 16'd15, 16'd14, 16'd15}; //
Append, Res H, Res V, Act H, Act V

    // Buffers for double buffering
    logic [23:0] buffer_color_output[0:1];
    logic [15:0] buffer_address_output[0:1];
    logic buffer_valid[0:1];
    logic [111:0] buffer_state[0:1];
    logic buffer_select = 1'b0;

    // Decode writedata fields
    logic [5:0] component;
    logic [3:0] action;
    logic [2:0] action_type;
    logic [12:0] action_data;
    logic buffer_toggle;

    assign component = writedata[31:26];
    assign action = writedata[20:17];
    assign action_type = writedata[16:14];
    assign buffer_toggle = writedata[13];
    assign action_data = writedata[12:0];

    // Address calculators for each buffer
    addr_cal addr_cal_ping(
        .pattern_info(buffer_state[0][111:32]),
        .sprite_info(buffer_state[0][31:0]),
        .hcount(hcount),
        .vcount(vcount),
        .addr_output(buffer_address_output[0]),
        .valid(buffer_valid[0])
    );

    addr_cal addr_cal_pong(
        .pattern_info(buffer_state[1][111:32]),
        .sprite_info(buffer_state[1][31:0]),
        .hcount(hcount),
        .vcount(vcount),
        .addr_output(buffer_address_output[1]),
        .valid(buffer_valid[1])
    );

    // Process input messages to control sprite parameters
    always_ff @(posedge clk) begin
        if (reset) begin
            buffer_select <= 0;
            buffer_state[0] <= 0;
            buffer_state[1] <= 0;
        end else begin
            case (action)
                4'b1111: begin  // Reset and toggle buffer
                    buffer_select <= buffer_toggle;
                    buffer_state[~buffer_toggle] <= 1'b0; // Clear
inactive buffer
                end
                4'h0001: if (component == COMPONENT_ID) begin
                    // Update buffer state based on input type
                    case (action_type)
                        3'b001: begin  // Set visibility and pattern
                            buffer_state[buffer_toggle][31:30] <=
{action_data[12], action_data[11]};
                            if (action_data[4:0] == 0) // Check for valid
pattern index
                                buffer_state[buffer_toggle][111:32] <=
pattern_table[action_data[4:0]];;
                        end
                        3'b010: buffer_state[buffer_toggle][29:20] <=
action_data[9:0]; // X position
                        3'b011: buffer_state[buffer_toggle][19:10] <=
action_data[9:0]; // Y position
                        3'b100: buffer_state[buffer_toggle][9:0] <=
action_data[9:0]; // Additional attributes (if any)
                    endcase
                end
            endcase
        end
    end

    // Determine RGB output based on active buffer state and validity
    always_comb begin
        if (buffer_valid[buffer_select] &&
buffer_address_output[buffer_select] < addr_limit) begin
            RGB_output =
color_palette[mem[buffer_address_output[buffer_select]]]; // Fetch color
using right-shift for 2-bit color index
        end else begin
            RGB_output = 24'h202020; // Default background color
        end
    end

    // Initialize pixel data from memory
    initial begin

$readmemh("/user/stud/fall21/bk2746/Projects/EmbeddedLab/Project_hw/on_chi
p_mem/Fireball.txt", mem);
```

```
    end
endmodule
```

Goomba Display:

```
/*
 * Avalon memory-mapped peripheral that generates VGA
 *
 * Stephen A. Edwards
 * Columbia University
 */

module Goomba_display (
    input logic        clk,
    input logic        reset,
    input logic [31:0] writedata,
    input logic [9:0]  hcount,
    input logic [9:0]  vcount,
    output logic [23:0] RGB_output
);

    // Constants for configuration
    parameter [5:0] COMPONENT_ID = 6'b000101;  // 5
    parameter [4:0] MAX_PATTERN_COUNT = 5'd_2;  // Number of patterns
supported
    parameter [15:0] ADDRESS_LIMIT = 16'd_384;  // Memory address limit
    parameter [4:0] CHILD_COMPONENT_LIMIT = 5'd_2;  // Limit for child
components
    logic [3:0] pixel_data [0:191];  // Pixel memory array
    logic [23:0] color_palette [0:3];  // Palette of colors
    logic [79:0] pattern_data [0:1];  // Pattern data

    // Define pattern and color data
    assign pattern_data[0] = {16'd_0, 16'd_16, 16'd_16, 16'd_16, 16'd_16};
    assign pattern_data[1] = {16'd_256, 16'd_16, 16'd_8, 16'd_16, 16'd_8};

    assign color_palette[0] = 24'h202020;  // Background color
    assign color_palette[1] = 24'h9c9c9c;  // Brown
    assign color_palette[2] = 24'h000000;  // Black
    assign color_palette[3] = 24'hffcec5;  // Peach
    // Buffers for rendering logic
    logic [23:0] buffer_color_output[0:1][0:1];
    logic [15:0] buffer_address_output[0:1][0:1];
    logic        buffer_valid[0:1][0:1];
    logic [111:0] buffer_state[0:1][0:1];
    logic        buffer_select = 1'b0;

    // Decode writedata fields
    logic [5:0] component;
    logic [4:0] child_component;
    logic [3:0] action;
    logic [2:0] action_type;
    logic [12:0] action_data;
    logic        buffer_toggle;

    // Decode input writedata
    assign component = writedata[31:26];
    assign child_component = writedata[25:21];
    assign action = writedata[20:17];
    assign action_type = writedata[16:14];
    assign buffer_toggle = writedata[13];
    assign action_data = writedata[12:0];

    // Address calculators for each child component in both buffer states
    genvar i;
    generate
        for (i = 0; i < CHILD_COMPONENT_LIMIT; i = i + 1) begin :
gen_addr_cal
            addr_cal address_calculator_ping(
                .pattern_info(buffer_state[0][i][111:32]),
                .sprite_info(buffer_state[0][i][31:0]),
                .hcount(hcount),
                .vcount(vcount),
                .addr_output(buffer_address_output[0][i]),
                .valid(buffer_valid[0][i])
            );

            addr_cal address_calculator_pong(
                .pattern_info(buffer_state[1][i][111:32]),
                .sprite_info(buffer_state[1][i][31:0]),
                .hcount(hcount),
                .vcount(vcount),
                .addr_output(buffer_address_output[1][i]),
                .valid(buffer_valid[1][i])
            );
        end
    endgenerate

    // Processing inputs and managing state
    always_ff @(posedge clk) begin
        case (action)
            4'b1111: begin  // Reset buffers based on toggle
                buffer_select = buffer_toggle;
                for (int j = 0; j < CHILD_COMPONENT_LIMIT; j = j + 1)
begin
                    buffer_state[~buffer_toggle][j][31] = 1'b0;  // Clear
visibility
                end
            end
            4'h0001: if (component == COMPONENT_ID && child_component <
CHILD_COMPONENT_LIMIT) begin
                // Update specific child component settings
                case (action_type)
                    3'b001: begin  // Visibility and pattern
                        buffer_state[buffer_toggle][child_component][31] =
action_data[12];
                        buffer_state[buffer_toggle][child_component][30] =
action_data[11];
                        if (action_data[4:0] < MAX_PATTERN_COUNT)

buffer_state[buffer_toggle][child_component][111:32] =
pattern_data[action_data[4:0]];
                    end
                    3'b010:
buffer_state[buffer_toggle][child_component][29:20] = action_data[9:0];
// X position
                    3'b011:
buffer_state[buffer_toggle][child_component][19:10] = action_data[9:0];
// Y position
```

```
                    3'b100:
buffer_state[buffer_toggle][child_component][9:0] = action_data[9:0];
// Additional attribute
                endcase
            end
        endcase
    end

    // Determine RGB output based on active buffer state and validity
    always_comb begin
        RGB_output = 24'h202020;  // Default to background color
        for (int k = 0; k < CHILD_COMPONENT_LIMIT; k = k + 1) begin
            if (buffer_valid[buffer_select][k]) begin
                // Directly use the address in the conditional expression
without declaring a new logic variable
                if (buffer_address_output[buffer_select][k] <
ADDRESS_LIMIT) begin
                    RGB_output =
color_palette[pixel_data[buffer_address_output[buffer_select][k] >> 1]];
                    if (RGB_output != 24'h202020) begin
                        break;  // Exit loop on first valid color
                    end
                end else begin
                    RGB_output = color_palette[0]; // Use default color if
address is out of limit
                end
            end
        end
    end

    // Initialize pixel data from memory
    initial begin

$readmemh("/user/stud/fall21/bk2746/Projects/EmbeddedLab/Project_hw/on_chi
p_mem/Goomba_2bit.txt", pixel_data);
    end

endmodule
```

Ground Display:

```
/*
 * Avalon memory-mapped peripheral that generates VGA
 *
 * Stephen A. Edwards
 * Columbia University
 */

module Ground_display (input logic        clk,
        input logic          reset,
        input logic [31:0]  writedata,
        input logic [9:0]   hcount,
        input logic [9:0]   vcount,

        output logic [23:0] RGB_output);


    parameter [5:0] COMPONENT_ID = 6'b001110; // 14
    parameter [4:0] pattern_num = 5'd_1;
    parameter [15:0] addr_limit = 16'd_256;
    logic [3:0] mem [0:127];
    logic [23:0] color_plate [0:3];
    logic [79:0] pattern_table [0:1];

    assign pattern_table[0] = {16'd_0, 16'd_16, 16'd_16, 16'd_650,
16'd_32};

    assign color_plate[0] = 24'h202020;
    assign color_plate[1] = 24'hFFFFFF;
    assign color_plate[2] = 24'h808080;
    assign color_plate[3] = 24'hD3D3D3;

    parameter [9:0] ground_height = 10'd_368;
    parameter [9:0] ceiling_height = 10'd_0;

    logic [23:0] buffer_RGB_out[0:3];
    logic [15:0] buffer_addr_out[0:3];
    logic        buffer_addr_valid[0:3];
    logic [111:0] frame_buffer_state[0:3];
    logic        buffer_select = 1'b0;
    logic [5:0] sub_comp;
    logic [4:0] child_comp;
    logic [3:0] info;
    logic [2:0] input_type;
    logic [12:0] input_msg;
    logic        buffer_state;

    assign sub_comp = writedata[31:26];
    assign child_comp = writedata[25:21];
    assign info = writedata[20:17];
    assign input_type = writedata[16:14];
    assign buffer_state = writedata[13];
    assign input_msg = writedata[12:0];

    assign frame_buffer_state[0][19:10] = ground_height;
    assign frame_buffer_state[1][19:10] = ground_height;
    assign frame_buffer_state[2][19:10] = ceiling_height;
    assign frame_buffer_state[3][19:10] = ceiling_height;

    logic [9:0] l_edge = 10'd0;
    logic [9:0] r_edge = 10'd0;

    addr_cal AC_ping_0(.pattern_info(frame_buffer_state[0][111:32]),
.sprite_info(frame_buffer_state[0][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_out[0]),
.valid(buffer_addr_valid[0]));
    addr_cal AC_pong_0(.pattern_info(frame_buffer_state[1][111:32]),
.sprite_info(frame_buffer_state[1][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_out[1]),
.valid(buffer_addr_valid[1]));
    addr_cal AC_ping_1(.pattern_info(frame_buffer_state[2][111:32]),
.sprite_info(frame_buffer_state[2][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_out[2]),
.valid(buffer_addr_valid[2]));
    addr_cal AC_pong_1(.pattern_info(frame_buffer_state[3][111:32]),
.sprite_info(frame_buffer_state[3][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_out[3]),
.valid(buffer_addr_valid[3]));
```

```
    always_ff @(posedge clk) begin
        case (info)

            4'b1111: begin
                buffer_select = buffer_state;
                if (buffer_state) begin
                    frame_buffer_state[0][31] = 1'b0;
                    frame_buffer_state[2][31] = 1'b0;
                end else begin
                    frame_buffer_state[1][31] = 1'b0;
                    frame_buffer_state[3][31] = 1'b0;
                end
            end

            4'h0001 : begin

                if (sub_comp == COMPONENT_ID) begin
                    case (input_type)
                        3'b001: begin

                            if (buffer_state) begin
                                frame_buffer_state[1][31] = input_msg[12];
                                frame_buffer_state[1][30] = input_msg[11];
                                frame_buffer_state[3][31] = input_msg[12];
                                frame_buffer_state[3][30] = input_msg[11];
                                if (input_msg[4:0] < pattern_num) begin
                                    frame_buffer_state[1][111:32] =
pattern_table[input_msg[4:0]];
                                    frame_buffer_state[3][111:32] =
pattern_table[input_msg[4:0]];
                                end
                            end else begin
                                frame_buffer_state[0][31] = input_msg[12];
                                frame_buffer_state[0][30] = input_msg[11];
                                frame_buffer_state[2][31] = input_msg[12];
                                frame_buffer_state[2][30] = input_msg[11];
                                if (input_msg[4:0] < pattern_num) begin
                                    frame_buffer_state[0][111:32] =
pattern_table[input_msg[4:0]];
```

```
                                    frame_buffer_state[2][111:32] =
pattern_table[input_msg[4:0]];
                                end
                            end
                        end
                        3'b010: begin
                            // x_coordinate
                            if (buffer_state) begin
                                frame_buffer_state[1][29:20] =
input_msg[9:0];
                                frame_buffer_state[3][29:20] =
input_msg[9:0];
                            end else begin
                                frame_buffer_state[0][29:20] =
input_msg[9:0];
                                frame_buffer_state[2][29:20] =
input_msg[9:0];
                            end
                        end
                        3'b011: begin

                            l_edge = input_msg[9:0];
                        end
                        3'b100: begin

                            r_edge = input_msg[9:0];
                        end
                    endcase
                end
            end
        endcase
    end

    assign buffer_RGB_out[0] =  (buffer_addr_out[0] < addr_limit)?
            (
                (buffer_addr_out[0][0])?
                    color_plate[mem[(buffer_addr_out[0][15:1])][3:2]] :
                    color_plate[mem[(buffer_addr_out[0][15:1])][1:0]]
            ) :
            color_plate[mem[0]];
    assign buffer_RGB_out[1] =  (buffer_addr_out[1] < addr_limit)?
            (
                (buffer_addr_out[1][0])?
                    color_plate[mem[(buffer_addr_out[1][15:1])][3:2]] :
                    color_plate[mem[(buffer_addr_out[1][15:1])][1:0]]
            ) :
            color_plate[mem[0]];

    assign buffer_RGB_out[2] =  (buffer_addr_out[2] < addr_limit)?
            (
                (buffer_addr_out[0][0])?
                    color_plate[mem[(buffer_addr_out[2][15:1])][3:2]] :
                    color_plate[mem[(buffer_addr_out[2][15:1])][1:0]]
            ) :
            color_plate[mem[0]];

    assign buffer_RGB_out[3] =  (buffer_addr_out[3] < addr_limit)?
            (
                (buffer_addr_out[1][0])?
                    color_plate[mem[(buffer_addr_out[3][15:1])][3:2]] :
                    color_plate[mem[(buffer_addr_out[3][15:1])][1:0]]
            ) :
            color_plate[mem[0]];

    always_comb begin
        if (vcount >= ground_height && vcount < ground_height + 32 &&
(hcount < l_edge || hcount > r_edge)) begin

            RGB_output = buffer_addr_valid[buffer_select ? 1 : 0] ?
                        buffer_RGB_out[buffer_select ? 1 : 0] :
                        24'h202020;
        end else if (vcount <= ceiling_height + 32 && vcount >
ceiling_height) begin

            RGB_output = buffer_addr_valid[buffer_select ? 3 : 2] ?
                        buffer_RGB_out[buffer_select ? 3 : 2] :
                        24'h202020;
        end else begin
            RGB_output = 24'h202020;
        end
    end


initial begin

$readmemh("/user/stud/fall21/bk2746/Projects/EmbeddedLab/Project_hw/on_chi
p_mem/Ground_2bit.txt", mem);
end


endmodule
```

Mario Display:

```
/*
 * Avalon memory-mapped peripheral that generates VGA
 *
 * Stephen A. Edwards
 * Columbia University
 */

module Mario_display (input logic        clk,
        input logic          reset,
        input logic [31:0]   writedata,
        input logic [9:0]    hcount,
        input logic [9:0]    vcount,

        output logic [23:0] RGB_output);

    parameter [5:0] COMPONENT_ID = 6'b000001; // 1
    parameter [4:0] pattern_num = 5'd_19;
    parameter [15:0] addr_limit = 16'd_7168;
    logic [3:0] mem [0:3583];
    logic [23:0] color_plate [0:3];
    logic [79:0] pattern_table [0:18];

    assign pattern_table[0] = {16'd_0, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
    assign pattern_table[1] = {16'd_256, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
```

```
    assign pattern_table[2] = {16'd_512, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
    assign pattern_table[3] = {16'd_768, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
    assign pattern_table[4] = {16'd_1024, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
    assign pattern_table[5] = {16'd_1280, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
    assign pattern_table[6] = {16'd_1536, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
    assign pattern_table[7] = {16'd_1792, 16'd_16, 16'd_32, 16'd_16,
16'd_32};
    assign pattern_table[8] = {16'd_2304, 16'd_16, 16'd_32, 16'd_16,
16'd_32};
    assign pattern_table[9] = {16'd_2816, 16'd_16, 16'd_32, 16'd_16,
16'd_32};
    assign pattern_table[10] = {16'd_3328, 16'd_16, 16'd_32, 16'd_16,
16'd_32};
    assign pattern_table[11] = {16'd_3840, 16'd_16, 16'd_32, 16'd_16,
16'd_32};
    assign pattern_table[12] = {16'd_4352, 16'd_16, 16'd_32, 16'd_16,
16'd_32};
    assign pattern_table[13] = {16'd_4864, 16'd_16, 16'd_24, 16'd_16,
16'd_24};
    assign pattern_table[14] = {16'd_5248, 16'd_16, 16'd_24, 16'd_16,
16'd_24};
    assign pattern_table[15] = {16'd_5632, 16'd_16, 16'd_32, 16'd_16,
16'd_32};
    assign pattern_table[16] = {16'd_6144, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
    assign pattern_table[17] = {16'd_6400, 16'd_16, 16'd_16, 16'd_16,
16'd_16};
    assign pattern_table[18] = {16'd_6656, 16'd_16, 16'd_32, 16'd_16,
16'd_32};

    assign color_plate[0] = 24'h202020;
    assign color_plate[1] = 24'hb53120;
    assign color_plate[2] = 24'h6b6d00;
    assign color_plate[3] = 24'hea9e22;
    logic [23:0] buffer_RGB_output[0:1];
    logic [15:0] buffer_addr_output[0:1];
    logic        buffer_addr_out_valid[0:1];
    logic [111:0] buffer_state_holder[0:1];
    logic        buffer_select = 1'b0;

    logic [5:0] sub_comp;
    logic [4:0] child_comp;
    logic [3:0] info;
    logic [2:0] input_type;
    logic [12:0] input_msg;
    logic       buffer_toggle;

    assign sub_comp = writedata[31:26];
    assign child_comp = writedata[25:21];
    assign info = writedata[20:17];
    assign input_type = writedata[16:14];
    assign buffer_toggle = writedata[13];
    assign input_msg = writedata[12:0];

    addr_cal AC_ping_0(.pattern_info(buffer_state_holder[0][111:32]),
.sprite_info(buffer_state_holder[0][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[0]),
.valid(buffer_addr_out_valid[0]));
    addr_cal AC_pong_0(.pattern_info(buffer_state_holder[1][111:32]),
.sprite_info(buffer_state_holder[1][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[1]),
.valid(buffer_addr_out_valid[1]));

    always_ff @(posedge clk) begin
        case (info)

            4'b1111: begin
                buffer_select = buffer_toggle;
                buffer_state_holder[~buffer_toggle][31] = 1'b0;
            end

            4'h0001 : begin

                if (sub_comp == COMPONENT_ID) begin
```

```
                    case (input_type)
                        3'b001: begin
                            // visible
                            buffer_state_holder[buffer_toggle][31] =
input_msg[12];
                            // fliped
                            buffer_state_holder[buffer_toggle][30] =
input_msg[11];
                            // pattern code
                            if (input_msg[4:0] < pattern_num)
                                buffer_state_holder[buffer_toggle][111:32]
= pattern_table[input_msg[4:0]];
                        end
                        3'b010: begin
                            // x_coordinate
                            buffer_state_holder[buffer_toggle][29:20] =
input_msg[9:0];
                        end
                        3'b011: begin
                            // y_coordinate
                            buffer_state_holder[buffer_toggle][19:10] =
input_msg[9:0];
                        end
                        3'b100: begin
                            // shift_amount
                            buffer_state_holder[buffer_toggle][9:0] =
input_msg[9:0];
                        end
                    endcase
                end
            end
       endcase
    end

    always_comb begin
        buffer_RGB_output[0] =  (buffer_addr_output[0] < addr_limit)?
            (
                (buffer_addr_output[0][0])?
                    color_plate[mem[(buffer_addr_output[0][15:1])][3:2]] :
                    color_plate[mem[(buffer_addr_output[0][15:1])][1:0]]
            ) :
            color_plate[mem[0]];

        buffer_RGB_output[1] =  (buffer_addr_output[1] < addr_limit)?
            (
                (buffer_addr_output[1][0])?
                    color_plate[mem[(buffer_addr_output[1][15:1])][3:2]] :
                    color_plate[mem[(buffer_addr_output[1][15:1])][1:0]]
            ) :
            color_plate[mem[0]];

        RGB_output = buffer_addr_out_valid[buffer_select]?
buffer_RGB_output[buffer_select] : 24'h202020;
    end

initial begin

$readmemh("/user/stud/fall21/bk2746/Projects/EmbeddedLab/Project_hw/on_chi
p_mem/Mario_2bit.txt", mem);
end


endmodule
```

Tube Display:

```
/*
 * Avalon memory-mapped peripheral that generates VGA
 *
 * Stephen A. Edwards
 * Columbia University
 */

module Tube_display (input logic        clk,
        input logic         reset,
        input logic [31:0]  writedata,
        input logic [9:0]   hcount,
        input logic [9:0]   vcount,

        output logic [23:0] RGB_output);
```

```
    parameter [5:0] COMPONENT_ID = 6'b001111; //15
    parameter [4:0] pattern_num = 5'd_2;
    parameter [15:0] addr_limit = 16'd_576;
    parameter [4:0] child_limit = 5'd_2;
    logic [3:0] mem [0:287];
    logic [23:0] color_plate [0:3];
    logic [79:0] pattern_table [0:1];

    assign pattern_table[0] = {16'd_0, 16'd_32, 16'd_16, 16'd_32,
16'd_16};
    assign pattern_table[1] = {16'd_544, 16'd_32, 16'd_1, 16'd_32,
16'd_128};

    assign color_plate[0] = 24'h202020;
    assign color_plate[1] = 24'h000000;
    assign color_plate[2] = 24'h8c0000;
    assign color_plate[3] = 24'h100000;

    logic [23:0] buffer_RGB_output[0:1][0:1];
    logic [15:0] buffer_addr_output[0:1][0:1];
    logic        buffer_addr_out_valid[0:1][0:1];
    logic [111:0] buffer_stateholder[0:1][0:1];
    logic        buffer = 1'b0;

    logic [5:0] sub_comp;
    logic [4:0] child_comp;
    logic [3:0] info;
    logic [2:0] input_type;
    logic [12:0] input_msg;
    logic       buffer_select;

    assign sub_comp = writedata[31:26];
    assign child_comp = writedata[25:21];
    assign info = writedata[20:17];
    assign input_type = writedata[16:14];
    assign buffer_select = writedata[13];
    assign input_msg = writedata[12:0];

    integer i, j, k;
    addr_cal AC_ping_0(.pattern_info(buffer_stateholder[0][0][111:32]),
.sprite_info(buffer_stateholder[0][0][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[0][0]),
.valid(buffer_addr_out_valid[0][0]));
    addr_cal AC_ping_1(.pattern_info(buffer_stateholder[0][1][111:32]),
.sprite_info(buffer_stateholder[0][1][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[0][1]),
.valid(buffer_addr_out_valid[0][1]));

    addr_cal AC_pong_0(.pattern_info(buffer_stateholder[1][0][111:32]),
.sprite_info(buffer_stateholder[1][0][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[1][0]),
.valid(buffer_addr_out_valid[1][0]));
    addr_cal AC_pong_1(.pattern_info(buffer_stateholder[1][1][111:32]),
.sprite_info(buffer_stateholder[1][1][31:0]), .hcount(hcount),
.vcount(vcount), .addr_output(buffer_addr_output[1][1]),
.valid(buffer_addr_out_valid[1][1]));

    always_ff @(posedge clk) begin
        case (info)

            4'b1111: begin
                buffer = buffer_select;
                for (i = 0; i < child_limit; i = i + 1) begin
                    buffer_stateholder[~buffer_select][i][31] = 1'b0;
                end

            end

            4'h0001 : begin

                if (sub_comp == COMPONENT_ID) begin
                    if (child_comp < child_limit) begin
                        case (input_type)
                            3'b001: begin
                                // visible

buffer_stateholder[buffer_select][child_comp][31] = input_msg[12];
                                // flipped
```

```
buffer_stateholder[buffer_select][child_comp][30] = input_msg[11];
                                // pattern code
                                if (input_msg[4:0] < pattern_num)

buffer_stateholder[buffer_select][child_comp][111:32] =
pattern_table[input_msg[4:0]];
                            end
                            3'b010: begin
                                // x_coordinate

buffer_stateholder[buffer_select][child_comp][29:20] = input_msg[9:0];
                            end
                            3'b011: begin
                                // y_coordinate

buffer_stateholder[buffer_select][child_comp][19:10] = input_msg[9:0];
                            end
                            3'b100: begin
                                // shift_amount

buffer_stateholder[buffer_select][child_comp][9:0] = input_msg[9:0];
                            end
                        endcase
                    end
                end
            end
       endcase
    end

    always_comb begin
        for (j = 0; j < child_limit; j = j + 1) begin
            buffer_RGB_output[0][j] =  (buffer_addr_output[0][j] <
addr_limit)?
            (
                (buffer_addr_output[0][j][0])?

color_plate[mem[(buffer_addr_output[0][j][15:1])][3:2]] :

color_plate[mem[(buffer_addr_output[0][j][15:1])][1:0]]
```

```
            ) :
            color_plate[mem[0]];

            buffer_RGB_output[1][j] =  (buffer_addr_output[1][j] <
addr_limit)?
            (
                (buffer_addr_output[1][j][0])?

color_plate[mem[(buffer_addr_output[1][j][15:1])][3:2]] :

color_plate[mem[(buffer_addr_output[1][j][15:1])][1:0]]
            ) :
            color_plate[mem[0]];
        end

        RGB_output = 24'h202020;
        for (k = 0; k < child_limit; k = k + 1) begin
            if ((buffer_RGB_output[buffer][k] != 24'h202020) &&
buffer_addr_out_valid[buffer][k]) begin
                RGB_output = buffer_RGB_output[buffer][k];
                break;
            end
        end
    end

initial begin

$readmemh("/user/stud/fall21/bk2746/Projects/EmbeddedLab/Project_hw/on_chi
p_mem/Tube_2bit.txt", mem);
end


endmodule
```

Peach_display

```
module Peach_display (
    input logic         clk,
    input logic         reset,
    input logic [31:0] writedata,
    input logic [9:0]  hcount,
    input logic [9:0]  vcount,
    output logic [23:0] RGB_output
);

    parameter [5:0] COMPONENT_ID = 6'b001010;
    parameter [15:0] addr_limit = 16'd3001;
    logic [3:0] mem [0:3000];
    logic [23:0] color_palette [0:4];
    logic [79:0] pattern_table [0:0];


    assign color_palette[0] = 24'hFF0000;
    assign color_palette[1] = 24'hFFCC66;
    assign color_palette[2] = 24'hFFFFFF;
    assign color_palette[3] = 24'h202020;


    assign pattern_table[0] = {16'd0, 16'd32, 16'd40, 16'd32, 16'd40};


    logic [23:0] buffer_color_output[0:1];
    logic [15:0] buffer_address_output[0:1];
    logic buffer_valid[0:1];
    logic [111:0] buffer_state[0:1];
    logic buffer_select = 1'b0;


    logic [5:0] component;
    logic [3:0] action;
    logic [2:0] action_type;
    logic [12:0] action_data;
    logic buffer_toggle;

    assign component = writedata[31:26];
    assign action = writedata[20:17];
    assign action_type = writedata[16:14];
    assign buffer_toggle = writedata[13];
    assign action_data = writedata[12:0];
    addr_cal addr_cal_ping(
        .pattern_info(buffer_state[0][111:32]),
        .sprite_info(buffer_state[0][31:0]),
        .hcount(hcount),
        .vcount(vcount),
        .addr_output(buffer_address_output[0]),
        .valid(buffer_valid[0])
    );

    addr_cal addr_cal_pong(
        .pattern_info(buffer_state[1][111:32]),
        .sprite_info(buffer_state[1][31:0]),
        .hcount(hcount),
        .vcount(vcount),
        .addr_output(buffer_address_output[1]),
        .valid(buffer_valid[1])
    );


    always_ff @(posedge clk) begin
        if (reset) begin
            buffer_select <= 0;
            buffer_state[0] <= 0;
            buffer_state[1] <= 0;
        end else begin
            case (action)
                4'b1111: begin
                    buffer_select <= buffer_toggle;
                    buffer_state[~buffer_toggle] <= 1'b0;
                end
                4'h0001: if (component == COMPONENT_ID) begin

                    case (action_type)
                        3'b001: begin
                            buffer_state[buffer_toggle][31:30] <=
{action_data[12], action_data[11]};
                            if (action_data[4:0] == 0)
                                buffer_state[buffer_toggle][111:32] <=
pattern_table[action_data[4:0]];;
                        end
                        3'b010: buffer_state[buffer_toggle][29:20] <=
action_data[9:0];
                        3'b011: buffer_state[buffer_toggle][19:10] <=
action_data[9:0];
                        3'b100: buffer_state[buffer_toggle][9:0] <=
action_data[9:0];
                    endcase
                end
            endcase
        end
    end


    always_comb begin
        if (buffer_valid[buffer_select] &&
buffer_address_output[buffer_select] < addr_limit) begin
            RGB_output =
color_palette[mem[buffer_address_output[buffer_select]]];
        end else begin
            RGB_output = 24'h202020;
        end
    end


    initial begin

$readmemh("/user/stud/fall21/bk2746/Projects/EmbeddedLab/Project_hw/on_chi
p_mem/Peach.txt", mem);
    end
endmodule
```

Ppu

```
/*
 * Avalon memory-mapped peripheral that generates VGA
 *
 * Stephen A. Edwards
 * Columbia University
 */

module ppu (
    input logic        clk,
    input logic        reset,
    input logic [31:0]  writedata,
    input logic [9:0]   hcount,
    input logic [9:0]   vcount,
    output logic [23:0] RGB_output
);

    localparam [23:0] BACKGROUND_COLOR = 24'h202020;
    logic [23:0] RGB_list[0:7];

    Mario_display Mario_0(
        .clk(clk), .reset(reset), .writedata(writedata),
        .hcount(hcount), .vcount(vcount), .RGB_output(RGB_list[0])
    );
    Goomba_display Goomba_0(
        .clk(clk), .reset(reset), .writedata(writedata),
        .hcount(hcount), .vcount(vcount), .RGB_output(RGB_list[1])
    );
    Block_display Block_0(
        .clk(clk), .reset(reset), .writedata(writedata),
        .hcount(hcount), .vcount(vcount), .RGB_output(RGB_list[2])
    );
    Fireball_display Fireball_0(
        .clk(clk), .reset(reset), .writedata(writedata),
        .hcount(hcount), .vcount(vcount), .RGB_output(RGB_list[3])
    );
    Bowser_display Bowser_0(
        .clk(clk), .reset(reset), .writedata(writedata),
        .hcount(hcount), .vcount(vcount), .RGB_output(RGB_list[4])
    );
    Peach_display Peach_0(
        .clk(clk), .reset(reset), .writedata(writedata),
        .hcount(hcount), .vcount(vcount), .RGB_output(RGB_list[5])
    );
    Ground_display Ground_0(
        .clk(clk), .reset(reset), .writedata(writedata),
        .hcount(hcount), .vcount(vcount), .RGB_output(RGB_list[6])
    );
    Tube_display Tube_0(
        .clk(clk), .reset(reset), .writedata(writedata),
        .hcount(hcount), .vcount(vcount), .RGB_output(RGB_list[7])
    );

    always_comb begin
        RGB_output = BACKGROUND_COLOR;
        for (int i = 0; i < 8; i = i + 1) begin
            if (RGB_list[i] != BACKGROUND_COLOR) begin
                RGB_output = RGB_list[i];
                break;
            end
        end
    end
endmodule
```

Vga_ball.sv

```
/*
 * Avalon memory-mapped peripheral that generates VGA
 *
 * Stephen A. Edwards
 * Columbia University
 */

module vga_ball(
    input logic        clk,
    input logic        reset,
    input logic [31:0] writedata,
    input logic        write,
    input logic        chipselect,
    input logic [2:0]  address,

    input logic        left_chan_ready,
    input logic        right_chan_ready,

    output logic [15:0] sample_data_l,
    output logic        sample_valid_l,
    output logic [15:0] sample_data_r,
    output logic        sample_valid_r,
    output logic [7:0]  VGA_R, VGA_G, VGA_B,
    output logic        VGA_CLK, VGA_HS, VGA_VS,
    output logic        VGA_BLANK_n,
    output logic        VGA_SYNC_n
);

    logic [10:0] hcount;
    logic [9:0]  vcount;
    logic [31:0] ppu_info;
    logic [5:0] note_en;
    wire sound_out;

    vga_counters counters(.clk50(clk), .*);
    // AudioGenerator audio(.clk(clk), .reset(reset), .note_en(note_en),
.sound_out(sound_out));
    always_ff @(posedge clk) begin
        if (reset) begin
            ppu_info <= 32'd_0;
        end else if (chipselect && write) begin
            case (address)
                3'h0: ppu_info <= writedata;
            endcase
        end
    end

    // PPU for video output logic
    logic [23:0] PPU_out;
    ppu game_ppu(.clk(clk), .reset(reset), .writedata(ppu_info),
.hcount(hcount[10:1]),
                 .vcount(vcount), .RGB_output(PPU_out));

    // VGA signal generation
    always_comb begin
        {VGA_R, VGA_G, VGA_B} = (VGA_BLANK_n && hcount >= 160 && hcount <
1120 && vcount < 400) ? PPU_out : 24'h000000;
    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
```

Addr_cal.sv

```
/*
 * Avalon memory-mapped peripheral that generates VGA
 *
 * Stephen A. Edwards
 * Columbia University
 */

module addr_cal (
        input logic [79:0]  pattern_info,
        input logic [31:0]  sprite_info,
        input logic [9:0]   hcount,
        input logic [9:0]   vcount,

        output logic [15:0] addr_output,
        output logic        valid);

    // sprite param
    logic [9:0]   x_pos;
    logic [9:0]   y_pos;
    logic [9:0]   shift_amount;
    logic         flip;
    logic         visible;

    assign visible = sprite_info[31];
    assign flip = sprite_info[30];
    assign x_pos = sprite_info[29:20];
    assign y_pos = sprite_info[19:10];
    assign shift_amount = sprite_info[9:0];

    // pattern param
    logic [15:0]    pattern_append, pattern_res_h, pattern_res_v,
pattern_act_h, pattern_act_v;
    logic [15:0]    x_pos_16, y_pos_16;
    assign pattern_append = pattern_info[79:64];
    assign pattern_res_h = pattern_info[63:48];
    assign pattern_res_v = pattern_info[47:32];
    assign pattern_act_h = pattern_info[31:16];
    assign pattern_act_v = pattern_info[15:0];
    assign x_pos_16[9:0] = x_pos;
    assign y_pos_16[9:0] = y_pos;

    //Calculation param
    logic [15:0]    pattern_addr_x, pattern_addr_y;
    logic [15:0]    res_x, res_y;

    always_comb begin
        if (visible
            && vcount >= y_pos_16 && vcount < y_pos_16 + pattern_act_v
            && hcount >= x_pos_16 && hcount < x_pos_16 + pattern_act_h
            ) begin
            res_x = (hcount - x_pos_16) % pattern_res_h;
            res_y = (vcount - y_pos_16) % pattern_res_v;
            //  non filped
            if (flip == 1'b0)begin
                pattern_addr_y = res_y * pattern_res_h;
                pattern_addr_x = res_x;
            end
            // filped
            else begin
                pattern_addr_y = res_y * pattern_res_h;
                pattern_addr_x = pattern_res_h - 1'b_1 - res_x;
                // pattern_addr_x = pattern_res_h - res_x;
            end
            addr_output = pattern_append + pattern_addr_y +
pattern_addr_x;
            valid = 1'b1;
        end

    else if (visible
            && vcount >= y_pos_16 && vcount < y_pos_16 + pattern_act_v
            ) begin
            res_x = 16'd0;
            res_y = (vcount - y_pos_16) % pattern_res_v;
            pattern_addr_y = res_y * pattern_res_h;
            pattern_addr_x = res_x;
            addr_output = pattern_append + pattern_addr_y +
pattern_addr_x;
            valid = 1'b0;
        end

        else begin
            addr_output = 16'd0;
            valid = 1'b0;
            pattern_addr_x = 16'd0;
            pattern_addr_y = 16'd0;
            res_x = 16'd0;
            res_y = 16'd0;
        end
    end


endmodule
```

Audio

```
module NoteGenerator (
    input wire clk,
    input wire reset,
    input wire note_enable,
    input integer half_period,
    output reg note_out
);
    integer counter = 0;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            counter <= 0;
            note_out <= 0;
        end else if (note_enable) begin
            if (counter >= half_period) begin
                note_out <= ~note_out; // Toggling output to create the
square wave
                counter <= 0;
            end else begin
                counter <= counter + 1;
            end
        end else begin
            note_out <= 0;
            counter <= 0;
        end
    end
endmodule
```

```verilog
module I2S_Controller(
    input wire clk,
    input wire reset,
    input wire [15:0] audio_data,
    output reg bclk,
    output reg ws,
    output reg sd,
    output reg ready
);
    localparam DIVIDER = 16;
    reg [7:0] bit_count = 0;
    reg [15:0] shift_reg = 0;
    integer clk_count = 0;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            clk_count <= 0;
            bclk <= 0;
            ws <= 0;
            bit_count <= 0;
            sd <= 0;
            ready <= 1;
        end else begin
            if (clk_count >= DIVIDER/2 - 1) begin
                clk_count <= 0;
                bclk <= ~bclk;
                if (bclk) begin
                    if (bit_count < 16) begin
                        sd <= shift_reg[15];
                        shift_reg <= shift_reg << 1;
                        bit_count <= bit_count + 1;
                    end else begin
                        bit_count <= 0;
                        ws <= ~ws;
                        ready <= 1;
                    end
                end
            end else begin
                clk_count <= clk_count + 1;
            end
        end
    end

    always @(posedge ws) begin
        shift_reg <= audio_data;
    end
endmodule

module AudioGenerator (
    input wire clk,
    input wire reset,
    output wire bclk,
    output wire ws,
    output wire sd
);
    localparam integer CLK_FREQ = 25000000; // 25 MHz clock
    localparam integer HP_C = CLK_FREQ / (2 * 262);
    localparam integer HP_D = CLK_FREQ / (2 * 294);
    localparam integer HP_E = CLK_FREQ / (2 * 330);
    localparam integer HP_F = CLK_FREQ / (2 * 349);
    localparam integer HP_G = CLK_FREQ / (2 * 392);
    localparam integer HP_A = CLK_FREQ / (2 * 440);
    localparam integer HP_B = CLK_FREQ / (2 * 494);
    localparam integer NOTE_DURATION = CLK_FREQ / 4; // Shorten note
duration for DEMO

    localparam TOTAL_NOTES = 16;
    integer note_sequence[TOTAL_NOTES] = {E, E, E, C, E, G, G, C, G, E, A,
B, B, A, G, E};
    integer durations[TOTAL_NOTES] = {NOTE_DURATION, NOTE_DURATION,
NOTE_DURATION, NOTE_DURATION, NOTE_DURATION, NOTE_DURATION, NOTE_DURATION,
NOTE_DURATION, NOTE_DURATION, NOTE_DURATION, NOTE_DURATION, NOTE_DURATION,
NOTE_DURATION, NOTE_DURATION, NOTE_DURATION, NOTE_DURATION};

    reg [5:0] note_en = 0;
    wire [5:0] note_out;
    reg [31:0] note_timer = 0;
    integer current_note = 0;

    // Instantiate note generators
    NoteGenerator gen_c(.clk(clk), .reset(reset),
.note_enable(note_en[C]), .half_period(HP_C), .note_out(note_out[C]));
    NoteGenerator gen_d(.clk(clk), .reset(reset),
.note_enable(note_en[D]), .half_period(HP_D), .note_out(note_out[D]));
    NoteGenerator gen_e(.clk(clk), .reset(reset),
.note_enable(note_en[E]), .half_period(HP_E), .note_out(note_out[E]));
    NoteGenerator gen_f(.clk(clk), .reset(reset),
.note_enable(note_en[F]), .half_period(HP_F), .note_out(note_out[F]));
    NoteGenerator gen_g(.clk(clk), .reset(reset),
.note_enable(note_en[G]), .half_period(HP_G), .note_out(note_out[G]));
    NoteGenerator gen_a(.clk(clk), .reset(reset),
.note_enable(note_en[A]), .half_period(HP_A), .note_out(note_out[A]));
    NoteGenerator gen_b(.clk(clk), .reset(reset),
.note_enable(note_en[B]), .half_period(HP_B), .note_out(note_out[B]));

    // Control logic for playing the song
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            current_note <= 0;
            note_timer <= 0;
            note_en <= 0;
        end else begin
            if (note_timer < durations[current_note]) begin
                note_timer <= note_timer + 1;
                note_en <= 1 << note_sequence[current_note];
            end else begin
                note_timer <= 0;
                current_note <= (current_note + 1) % TOTAL_NOTES;
            end
        end
    end

    // Combine outputs to a single 16-bit sound output ==> stereo
duplication
    wire [15:0] sound_data = {note_out, note_out};

    // Instantiate I2S Controller
    I2S_Controller i2s(
        .clk(clk),
        .reset(reset),
        .audio_data(sound_data),
        .bclk(bclk),
        .ws(ws),
        .sd(sd),
        .ready()
    );
endmodule

// RECALL:
// Gotta assign the bclk, ws, and sd signals to the FPGA pins that are
connected to the WM8731 codec on FPGA
// TO--DO: modifying the Quartus project's pin assignments file (.qsf)
```

Code

SuperMario.c

```
/*
 * Game Logic
 * Edited on April 22 2024 by Brandon Khadan
*/

#include <stdio.h>
#include <math.h>
#include "vga_ball.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdint.h>
#include <time.h>
#include "usbkeyboard.h"
#include <pthread.h>
#include "game_struct.h"
#include "game_animation.h"

int vga_ball_fd;
```

```c
int block_index;

int info_001 = 1;
int info_010 = 2;
int info_011 = 3;
int info_100 = 4;

int frame_counter = 0;

int can_jump = 0;

int bowser_alive = 1;
int lives = 2;
float speed = 0.5;

struct libusb_device_handle *keyboard;
enum key_input{KEY_NONE, KEY_JUMP, KEY_LEFT, KEY_RIGHT, KEY_NEWGAME,
KEY_END};
enum key_input current_key;
uint8_t endpoint_address;
pthread_t input_thread;

void write_to_hardware(int vga_fd, int register_address, int data) {
    vga_ball_arg_t vla;
    vla.addr = register_address;
    vla.info = data;

    if (ioctl(vga_fd, VGA_BALL_WRITE_BACKGROUND, &vla) < 0) {
        fprintf(stderr, "Failed to write data to hardware\n");
    }
}

void flush_mario(const Entity *entity, int frame_select) {

    int visible = entity->render.visible;
    int flip = entity->render.flip;
    int x = entity->position.x;
    int y = entity->position.y;
    int pattern_code = entity->render.pattern_code;
    if (frame_counter % 100 == 0)
        printf("Flushing MARIO - Visible: %d, Flip: %d, X: %d, Y: %d, 
Pattern: %d\n", visible, flip, x, y, pattern_code);

    write_to_hardware(vga_ball_fd, 0, (int)((1 << 26) + (1 << 17) + 
(info_001 << 14) + (frame_select << 13) + (visible << 12) + (flip << 11) + 
(pattern_code & 0x1F)));
    write_to_hardware(vga_ball_fd, 0, (int)((1 << 26) + (1 << 17) + 
(info_010 << 14) + (frame_select << 13) + (x & 0x3FF)));
    write_to_hardware(vga_ball_fd, 0, (int)((1 << 26) + (1 << 17) + 
(info_011 << 14) + (frame_select << 13) + (y & 0x3FF)));
}

void flush_goomba(const Entity *entity, int frame_select) {

    int visible = entity->render.visible;
    int flip = entity->render.flip;
    int x = entity->position.x;
    int y = entity->position.y;
    int pattern_code = entity->render.pattern_code;
    if (frame_counter % 100 == 0)
        printf("Flushing GOOMBA - Visible: %d, Flip: %d, X: %d, Y: %d, 
Pattern: %d\n", visible, flip, x, y, pattern_code);

    write_to_hardware(vga_ball_fd, 0, (int)((5 << 26) + (1 << 17) + 
(info_001 << 14) + (frame_select << 13) + (visible << 12) + (flip << 11) + 
(pattern_code & 0x1F)));
    write_to_hardware(vga_ball_fd, 0, (int)((5 << 26) + (1 << 17) + 
(info_010 << 14) + (frame_select << 13) + (x & 0x3FF)));
    write_to_hardware(vga_ball_fd, 0, (int)((5 << 26) + (1 << 17) + 
(info_011 << 14) + (frame_select << 13) + (y & 0x3FF)));
}

void flush_block(const Entity *entity, int frame_select) {

    int visible = entity->render.visible;
    int flip = entity->render.flip;
    int x = entity->position.x;
    int y = entity->position.y;
    int pattern_code = entity->render.pattern_code;
    int entityTypeCode = entity->state.type;

    if (frame_counter % 100 == 0)
        printf("Flushing BLOCK - Type: %d, Visible: %d, Flip: %d, X: %d,
Y: %d, Pattern: %d\n", entityTypeCode, visible, flip, x, y, pattern_code);

    write_to_hardware(vga_ball_fd, 0, (int)((7 << 26) + ((block_index &
0x1F) << 21) + (1 << 17) + (info_001 << 14) + (frame_select << 13) +
(visible << 12) + (0 << 11) + (pattern_code & 0x1F)));
    write_to_hardware(vga_ball_fd, 0, (int)((7 << 26) + ((block_index &
0x1F) << 21) + (1 << 17) + (info_010 << 14) + (frame_select << 13) + (x &
0x3FF)));
    write_to_hardware(vga_ball_fd, 0, (int)((7 << 26) + ((block_index &
0x1F) << 21) + (1 << 17) + (info_011 << 14) + (frame_select << 13) + (y &
0x3FF)));
    block_index += 1;
}

void flush_peach(const Entity *entity, int frame_select) {
    int visible = entity->render.visible;
    int flip = entity->render.flip;
    int x = entity->position.x;
    int y = entity->position.y;
    int pattern_code = entity->render.pattern_code;

    if (frame_counter % 100 == 0)
        printf("Flushing Peach - Visible: %d, Flip: %d, X: %d, Y: %d,
Pattern: %d\n", visible, flip, x, y, pattern_code);

    write_to_hardware(vga_ball_fd, 0, (int)((10 << 26) + (1 << 17) +
(info_001 << 14) + (frame_select << 13) + (visible << 12) + (flip << 11) +
(pattern_code & 0x1F)));
    write_to_hardware(vga_ball_fd, 0, (int)((10 << 26) + (1 << 17) +
(info_010 << 14) + (frame_select << 13) + (x & 0x3FF)));
    write_to_hardware(vga_ball_fd, 0, (int)((10 << 26) + (1 << 17) +
(info_011 << 14) + (frame_select << 13) + (y & 0x3FF)));
}

void flush_fireball(const Entity *entity, int frame_select) {
    int visible = entity->render.visible;
    int flip = entity->render.flip;
    int x = entity->position.x;
    int y = entity->position.y;
    int pattern_code = entity->render.pattern_code;

    if (frame_counter % 100 == 0)
        printf("Flushing Fireball - Visible: %d, Flip: %d, X: %d, Y: %d,
Pattern: %d\n", visible, flip, x, y, pattern_code);

    write_to_hardware(vga_ball_fd, 0, (int)((8 << 26) + (1 << 17) +
(info_001 << 14) + (frame_select << 13) + (visible << 12) + (flip << 11) +
(pattern_code & 0x1F)));
    write_to_hardware(vga_ball_fd, 0, (int)((8 << 26) + (1 << 17) +
(info_010 << 14) + (frame_select << 13) + (x & 0x3FF)));
    write_to_hardware(vga_ball_fd, 0, (int)((8 << 26) + (1 << 17) +
(info_011 << 14) + (frame_select << 13) + (y & 0x3FF)));
}


void flush_tube(const Entity *entity, int frame_select) {
    int visible = entity->render.visible;
    int flip = entity->render.flip;
    int x = entity->position.x;
    int y = entity->position.y;
    int pattern_code = entity->render.pattern_code;

    if (frame_counter % 100 == 0)
        printf("Flushing TUBE - Visible: %d, Flip: %d, X: %d, Y: %d,
Pattern: %d\n", visible, flip, x, y, pattern_code);

    write_to_hardware(vga_ball_fd, 0, (int)((15 << 26) + ((0&0x1F) << 21)
+ (1 << 17) + (info_001 << 14) + (frame_select << 13) + (1 << 12) + (flip
<< 11) + (ANI_TUBE_H & 0x1F)));
    write_to_hardware(vga_ball_fd, 0, (int)((15 << 26) + ((0&0x1F) << 21)
+ (1 << 17) + (info_010 << 14) + (frame_select << 13) + (x & 0x3FF)));
    write_to_hardware(vga_ball_fd, 0, (int)((15 << 26) + ((0&0x1F) << 21)
+ (1 << 17) + (info_011 << 14) + (frame_select << 13) + (y & 0x3FF)));
```

```
    write_to_hardware(vga_ball_fd, 0, (int)((15 << 26) + ((1&0x1F) << 21)
+ (1 << 17) + (info_001 << 14) + (frame_select << 13) + (1 << 12) + (flip
<< 11) + (ANI_TUBE_B & 0x1F)));
    write_to_hardware(vga_ball_fd, 0, (int)((15 << 26) + ((1&0x1F) << 21)
+ (1 << 17) + (info_010 << 14) + (frame_select << 13) + (x & 0x3FF)));
    write_to_hardware(vga_ball_fd, 0, (int)((15 << 26) + ((1&0x1F) << 21)
+ (1 << 17) + (info_011 << 14) + (frame_select << 13) + (y & 0x3FF)));
}

void flush_bowser(const Entity *entity, int frame_select) {
    int visible = entity->render.visible;
    int flip = entity->render.flip;
    int x = entity->position.x;
    int y = entity->position.y;
    int pattern_code = entity->render.pattern_code;

    if (frame_counter % 100 == 0)
        printf("Flushing Bowser - Visible: %d, Flip: %d, X: %d, Y: %d,
Pattern: %d\n", visible, flip, x, y, pattern_code);

    write_to_hardware(vga_ball_fd, 0, (int)((9 << 26) + (1 << 17) +
(info_001 << 14) + (frame_select << 13) + (visible << 12) + (flip << 11) +
(pattern_code & 0x1F)));
    write_to_hardware(vga_ball_fd, 0, (int)((9 << 26) + (1 << 17) +
(info_010 << 14) + (frame_select << 13) + (x & 0x3FF)));
    write_to_hardware(vga_ball_fd, 0, (int)((9 << 26) + (1 << 17) +
(info_011 << 14) + (frame_select << 13) + (y & 0x3FF)));
}

void flush_ground(Entity *entity, int camera_pos, int frame_select) {
    int visible = entity->render.visible;
    int flip = entity->render.flip;
    int x = entity->position.x;
    int y = entity->position.y;
    int pattern_code = entity->render.pattern_code;
    int left_edge = entity->position.x + entity->position.width;
    int right_edge = left_edge + GROUND_PIT_WIDTH;

    if (frame_counter % 100 == 0)
```

```
        printf("Flushing Ground - Visible: %d, Flip: %d, X: %d, Y: %d, 
Pattern: %d, l: %d. r:%d\n", visible, flip, x, y, pattern_code, left_edge, 
right_edge);

    write_to_hardware(vga_ball_fd, 0, (int)((14 << 26) + ((0&0x1F) << 21) 
+ (1 << 17) + (info_001 << 14) + (frame_select << 13) + (visible << 12) + 
(0 << 11) + (0 & 0x1F)));
    write_to_hardware(vga_ball_fd, 0, (int)((14 << 26) + ((0&0x1F) << 21) 
+ (1 << 17) + (info_010 << 14) + (frame_select << 13) + ((15 - 
(camera_pos%16)) & 0x3FF)));
    write_to_hardware(vga_ball_fd, 0, (int)((14 << 26) + ((0&0x1F) << 21) 
+ (1 << 17) + (info_011 << 14) + (frame_select << 13) + (0 & 0x3FF)));
    write_to_hardware(vga_ball_fd, 0, (int)((14 << 26) + ((0&0x1F) << 21) 
+ (1 << 17) + (info_100 << 14) + (frame_select << 13) + (0 & 0x3FF)));
}

void flush_entity(Entity *entity, int frame_select, int camera_pos) {
    if (entity->render.pattern_code > 6 || entity->render.pattern_code < 0
) return;
    if (entity->state.type > TYPE_EMP || entity->state.type < 0) return;
    if (entity->state.active != 1) return;

    switch (entity->state.type) {
        case TYPE_MARIO_SMALL:
        case TYPE_MARIO_LARGE:
            flush_mario(entity, frame_select);
            break;
        case TYPE_GOOMBA:
            flush_goomba(entity, frame_select);
            break;
        case TYPE_BLOCK_A:
        case TYPE_BLOCK_B_1:
        case TYPE_BLOCK_B_2:
        case TYPE_BLOCK_B_3:
        case TYPE_BLOCK_B_4:
        case TYPE_BLOCK_B_16:
        case TYPE_BLOCK_A_H_8:
        case TYPE_BLOCK_OBJ_C:
        case TYPE_BLOCK_OBJ_M:
            flush_block(entity, frame_select);
            break;
        case TYPE_TUBE:
            flush_tube(entity, frame_select);
            break;
        case TYPE_BOWSER:
            flush_bowser(entity,frame_select);
            break;
        case TYPE_GROUND:
            flush_ground(entity, camera_pos, frame_select);

            break;
        case TYPE_PEACH:
            flush_peach(entity, frame_select);
            break;
        case TYPE_FIREBALL:
            flush_fireball(entity, frame_select);
            break;
        default:
            break;
    }
}

void flush_frame(Game *game, int frame_select) {
    int entity_index;
    block_index = 0;
    Entity *entity;

    for (entity_index = 0; entity_index < MAX_ENTITIES; entity_index++) {
        entity = &game->entities[entity_index];

        if (entity->state.active == 1 && entity->render.visible == 1) {
            flush_entity(entity, frame_select, game->camera_pos);
        }
    }

    write_to_hardware(vga_ball_fd, 0, (int)((1 << 26) + (0xf << 17) +
(frame_select << 13)));

    frame_counter = (frame_counter >= FRAME_LIMIT) ? 0 : frame_counter +
1;
```

```
}

void *input_thread_function(void *ignored)
{
    struct usb_keyboard_packet packet;
    int transferred;
    int r;
    struct timeval timeout = { 0, 500000 };
    uint8_t first, second, chosen;

    for (;;) {
        r = libusb_interrupt_transfer(keyboard, endpoint_address,
(unsigned char *)&packet, sizeof(packet), &transferred, 0);
        if (r == 0 && transferred == sizeof(packet)) {

            first = packet.keycode[0];
            second = packet.keycode[1];
            chosen = 0;

            if (first != 0 && second != 0) {

                if (first == second) {
                    usleep(5000);
                    continue;
                } else {
                    chosen = second;
                }

            } else {
                chosen = first;
            }

            switch(chosen) {
                case 0x2C:
                    current_key = KEY_JUMP;
                    break;
                case 0x04:
                    current_key = KEY_LEFT;
                    break;
                case 0x07:
                    current_key = KEY_RIGHT;
                    break;
                case 0x0A:
                    current_key = KEY_NEWGAME;
                    break;
                default:
                    current_key = KEY_NONE;
                    break;
            }
        } else {
            if (r == LIBUSB_ERROR_NO_DEVICE) {

                fprintf(stderr, "Keyboard disconnected.\n");
                break;
            }

            fprintf(stderr, "Transfer error: %s\n", libusb_error_name(r));
            current_key = KEY_NONE;
            libusb_handle_events_timeout(NULL, &timeout);
        }
    }
    return NULL;
}

void handle_collision_with_block(Entity *mario, Entity *other, enum
contact type) {
    if (type == UP) {
        mario->motion.vy = 0;
    } else if (type == DOWN) {
        mario->motion.vy = 0;
        mario->position.y = other->position.y - mario->position.height;
        can_jump = 1;
    } else if (type == LEFT && mario->render.flip == 1) {
        mario->motion.vx = 0;
    } else if (type == RIGHT && mario->render.flip == 0) {
        mario->motion.vx = 0;
    }
}
```

```c
void handle_collision_with_tube(Entity *mario, Entity *other, enum contact
type) {
    if (type == LEFT && mario->render.flip == 1) {
        mario->motion.vx = 0;
    } else if(type == RIGHT && mario->render.flip == 0) {
        mario->motion.vx = 0;
    } else if (type == DOWN) {
        mario->motion.vy = 0;
        mario->position.y = other->position.y - mario->position.height;
        can_jump = 1;
    }
}

void handle_collision_with_ground(Entity *mario, Entity *other, enum
contact type) {
    if (type == DOWN) {
        mario->motion.vy = 0;
        mario->position.y = other->position.y - mario->position.height;
        can_jump = 1;
    }
}

void process_mario_logic(Entity *mario, Game *game) {
    if (mario == NULL) {
        printf("Mario entity is NULL\n");
        return;
    }

    // Apply gravity
    mario->motion.ay = GRAVITY;

    // Reset horizontal acceleration
    mario->motion.ax = 0;

    // Handle horizontal input
    if (current_key == KEY_LEFT) {
        mario->motion.ax = -WALK_ACC;
        mario->render.flip = 1; // Mario faces left
    } else if (current_key == KEY_RIGHT) {
        mario->motion.ax = WALK_ACC;
        mario->render.flip = 0; // Mario faces right
    }

    // Handle jump input
    if (current_key == KEY_JUMP && can_jump) {
        mario->motion.vy = -JUMP_INIT_V_LARGE;
        can_jump = 0;
    }

    // Apply friction if Mario is on the ground and moving
    if (mario->motion.vy == 0 && fabs(mario->motion.vx) > MOTION_MIN) {
        mario->motion.ax -= mario->motion.vx * FRICTION;
    }

    if (fabs(mario->motion.vx) < 0.08f) {
        mario->motion.vx = 0;
    }

    // Update velocities
    mario->motion.vx += mario->motion.ax;
    mario->motion.vy += mario->motion.ay;

    // Limit speeds
    mario->motion.vx = fminf(fmaxf(mario->motion.vx, -MAX_SPEED_H),
MAX_SPEED_H);
    mario->motion.vy = fminf(fmaxf(mario->motion.vy, -MAX_SPEED_V_JUMP),
MAX_SPEED_V);

    // Collision detection
    for (int i = 1; i < MAX_ENTITIES; i++) {
        Entity *other = &game->entities[i];
        if (other == NULL || !other->state.active) continue;
        enum contact contactType = hitbox_contact(mario, other);
        if (contactType != NONE) {
            switch (other->state.type) {
                case TYPE_GOOMBA:
                    break;
                case TYPE_BLOCK_A:
                case TYPE_BLOCK_B_1:
                case TYPE_BLOCK_B_2:
```

```
                case TYPE_BLOCK_B_3:
                case TYPE_BLOCK_B_4:
                case TYPE_BLOCK_B_16:
                case TYPE_BLOCK_A_H_8:
                case TYPE_BLOCK_OBJ_C:
                case TYPE_BLOCK_OBJ_M:
                    handle_collision_with_block(mario, other,
contactType);
                    break;
                case TYPE_TUBE:
                    handle_collision_with_tube(mario, other, contactType);
                    break;
                case TYPE_GROUND:
                    handle_collision_with_ground(mario, other,
contactType);
                    break;
                case TYPE_PEACH:
                    mario->state.state = STATE_DEAD;
                default:
                    break;
            }
        }
    }

    mario->position.x += mario->motion.vx;
    game->camera_velocity = 0;
    if (mario->position.x > ((2*CAMERA_SIZE)/3)) {
        mario->position.x = ((2*CAMERA_SIZE)/3) - 1;
        game->camera_velocity = mario->motion.vx;

        if (game->camera_pos < 355 || bowser_alive == 0)
            game->camera_pos += mario->motion.vx;
        if (game->camera_pos < game->camera_start) {
            game->camera_pos = game->camera_start;
        }
    } else if (mario->position.x < 70) {
        mario->position.x = 70 + 1;
    }

    mario->position.y += mario->motion.vy;
    if (mario->position.y > GROUND_LEVEL) {
        mario->state.state = STATE_DEAD;
        mario->state.active = 0;
    }
}

void process_goomba_logic(Entity *goomba, Game *game) {
    enum contact contactType;
    Entity *other;
    if (!goomba->state.active)
        return;

    goomba->motion.ay = GRAVITY;

    goomba->motion.vx = (goomba->render.flip == 0) ? -MAX_SPEED_H * 0.5 :
MAX_SPEED_H * 0.5;
    for (int i = 0; i < MAX_ENTITIES; i++) {
        other = &game->entities[i];
        if (other == NULL || !other->state.active || other == goomba)
continue;

        contactType = hitbox_contact(goomba, other);
        if (contactType != NONE) {
            switch (other->state.type) {
                case TYPE_MARIO_SMALL:
                    if (contactType == UP) {

                        goomba->state.state = STATE_DEAD;
                        goomba->state.active = 0;
                        other->motion.vy = -JUMP_INIT_V_SMALL;
                    } else {

                        other->state.state = STATE_DEAD;
                    }
                    break;
                case TYPE_MARIO_LARGE:
                    if (contactType == UP) {
                        goomba->state.state = STATE_DEAD;
                        goomba->state.active = 0;
                        other->motion.vy = -JUMP_INIT_V_SMALL;
                    } else {
                        other->state.type = TYPE_MARIO_SMALL;
                        other->state.state = STATE_HIT;
                    }
                    break;
                case TYPE_GROUND:
                    if (contactType == DOWN) {
                        goomba->motion.vy = 0;
                        goomba->position.y = other->position.y -
goomba->position.height;
                    }
                    break;
                case TYPE_TUBE:
                case TYPE_BLOCK_A:
                case TYPE_BLOCK_B_1:
                case TYPE_BLOCK_B_2:
                case TYPE_BLOCK_B_3:
                case TYPE_BLOCK_B_4:
                case TYPE_BLOCK_B_16:
                case TYPE_BLOCK_A_H_8:
                case TYPE_BLOCK_OBJ_C:
                case TYPE_BLOCK_OBJ_M:
                    if (contactType == LEFT || contactType == RIGHT) {
                        goomba->render.flip = (goomba->render.flip == 0) ?
1 : 0;
                        goomba->motion.vx = -goomba->motion.vx;
                    }
                    break;
            }
        }
    }

    goomba->motion.vy += goomba->motion.ay;
    goomba->position.x += goomba->motion.vx;
    goomba->position.y += goomba->motion.vy;
    goomba->motion.ax = 0;
    goomba->motion.ay = 0;

    if (goomba->position.y > GROUND_LEVEL) {
        goomba->state.state = STATE_DEAD;
        goomba->state.active = 0;
    }

    if (game->camera_pos < 355 || bowser_alive == 0) {
        goomba->position.x -= game->camera_velocity;
    }

    if (goomba->position.x < game->camera_start) {
        goomba->state.active = 0;
        printf("Cull Goomba");
    }

    // if(goomba->position.x > game->camera_start + CAMERA_SIZE) {
    //  goomba->render.visible = 0;
    // } else {
    //  goomba->render.visible = 1;
    // }
}

void process_bowser_logic(Entity *bowser, Game *game) {
    enum contact contactType;
    Entity *other;
    if (!bowser->state.active)
        return;

    bowser->motion.ay = GRAVITY;

    bowser->motion.vx = (bowser->render.flip == 0) ? -MAX_SPEED_H * speed
: MAX_SPEED_H * speed;
    for (int i = 0; i < MAX_ENTITIES; i++) {
        other = &game->entities[i];
        if (other == NULL || !other->state.active || other == bowser)
continue;

        contactType = hitbox_contact(bowser, other);
        if (contactType != NONE) {
            switch (other->state.type) {
                case TYPE_MARIO_SMALL:
                    if (contactType == UP) {
```

```
                    if (lives == 0) {
                        bowser->state.state = STATE_DEAD;
                        bowser->state.active = 0;
                        bowser->render.visible = 0;
                        bowser_alive = 0;
                    } else {
                        lives -= 1;
                        speed += 0.15;
                    }

                    other->motion.vy = -JUMP_INIT_V_LARGE;
                } else {

                    other->state.state = STATE_DEAD;
                }
                break;
            case TYPE_MARIO_LARGE:
                if (contactType == UP) {
                    bowser->state.state = STATE_DEAD;
                    bowser->state.active = 0;
                    other->motion.vy = -JUMP_INIT_V_SMALL;
                } else {
                    other->state.type = TYPE_MARIO_SMALL;
                    other->state.state = STATE_HIT;
                }
                break;
            case TYPE_GROUND:
                if (contactType == DOWN) {
                    bowser->motion.vy = 0;
                    bowser->position.y = other->position.y -
bowser->position.height;
                }
                break;
            case TYPE_TUBE:
            case TYPE_BLOCK_A:
            case TYPE_BLOCK_B_1:
            case TYPE_BLOCK_B_2:
            case TYPE_BLOCK_B_3:
            case TYPE_BLOCK_B_4:
            case TYPE_BLOCK_B_16:
                case TYPE_BLOCK_A_H_8:
                case TYPE_BLOCK_OBJ_C:
                case TYPE_BLOCK_OBJ_M:
                    if (contactType == LEFT || contactType == RIGHT) {
                        bowser->render.flip = (bowser->render.flip == 0) ?
1 : 0;
                        bowser->motion.vx = -bowser->motion.vx;
                    }
                    break;
            }
        }
    }

    bowser->motion.vy += bowser->motion.ay;
    bowser->position.x += bowser->motion.vx;
    bowser->position.y += bowser->motion.vy;
    bowser->motion.ax = 0;
    bowser->motion.ay = 0;

    if (bowser->position.y > GROUND_LEVEL) {
        // bowser->state.state = STATE_DEAD;
        // bowser->state.active = 0;
        // bowser->render.visible = 0;
        bowser->position.y = GROUND_LEVEL - 32;
    }

    if (game->camera_pos < 355 || bowser_alive == 0) {
        bowser->position.x -= game->camera_velocity;
    }

    if (bowser->position.x < game->camera_start) {
        bowser->state.active = 0;
        printf("Cull Goomba");
    }

    // if(bowser->position.x > game->camera_start + CAMERA_SIZE) {
    //  bowser->render.visible = 0;
    // } else {
    //  bowser->render.visible = 1;
    // }
```

```
}

void process_fireball_logic(Entity *fireball, Game *game) {
    enum contact contactType;
    Entity *other;
    if (!fireball->state.active)
        return;

    fireball->motion.ay = 0;

    fireball->motion.vx = (fireball->render.flip == 0) ? -MAX_SPEED_H :
MAX_SPEED_H;
    for (int i = 0; i < MAX_ENTITIES; i++) {
        other = &game->entities[i];
        if (other == NULL || !other->state.active || other == fireball)
continue;

        contactType = hitbox_contact(fireball, other);
        if (contactType != NONE) {
            switch (other->state.type) {
                case TYPE_MARIO_SMALL:
                case TYPE_MARIO_LARGE:
                    other->state.state = STATE_DEAD;
                    break;
            }
        }
    }

    fireball->motion.vy += fireball->motion.ay;
    fireball->position.x += fireball->motion.vx;
    fireball->position.y += fireball->motion.vy;
    fireball->motion.ax = 0;
    fireball->motion.ay = 0;

    if (fireball->position.y > GROUND_LEVEL) {
        fireball->state.state = STATE_DEAD;
        fireball->state.active = 0;
        fireball->render.visible = 0;
    }
    fireball->position.x -= game->camera_velocity;
    if (fireball->position.x < game->camera_start) {
        fireball->render.flip = !fireball->render.flip;
        fireball->position.x = game->camera_start + 15;
    } else if(fireball->position.x > game->camera_start + CAMERA_SIZE) {
        fireball->render.flip = !fireball->render.flip;
        fireball->position.x = game->camera_start + CAMERA_SIZE - 30;
    }

    if(fireball->position.x > game->camera_start + CAMERA_SIZE) {
        fireball->render.visible = 0;
    } else {
        fireball->render.visible = 1;
    }
}

int main() {
    pthread_t input_thread;
    Game game;
    const char *device_path = "/dev/vga_ball";
    int frame_select = 0;
    Entity *mario;
    Entity *entity;

    if ((vga_ball_fd = open(device_path, O_RDWR)) < 0) {
        fprintf(stderr, "Failed to open hardware device: %s\n",
device_path);
        return EXIT_FAILURE;
    }

    if ((keyboard = openkeyboard(&endpoint_address)) == NULL) {
        fprintf(stderr, "Did not find a keyboard\n");
        exit(EXIT_FAILURE);
    }

    if (pthread_create(&input_thread, NULL, input_thread_function, NULL)
!= 0) {
        fprintf(stderr, "Failed to create input thread\n");
        return EXIT_FAILURE;
    }
    new_game(&game);

    while (1) {

        mario = &game.entities[0];
        if (mario->state.state == STATE_DEAD || current_key ==
KEY_NEWGAME) {
            new_game(&game);
            lives = 2;
            bowser_alive = 1;
            speed = 0.5;
            continue;
        }

        for (int i = 0; i < MAX_ENTITIES; i++) {
            entity = &game.entities[i];

            if (!entity) continue;

            if (entity->state.active == 1) {
                switch (entity->state.type) {
                    case TYPE_MARIO_SMALL:
                    case TYPE_MARIO_LARGE:
                        process_mario_logic(entity, &game);
                        break;
                    case TYPE_GOOMBA:
                        process_goomba_logic(entity, &game);
                        break;
                    case TYPE_GROUND:
                        // entity->position.x -= game.camera_velocity;

                        // if (entity->position.x > game.camera_start +
CAMERA_SIZE) {
                        //  entity->render.visible = 0;
                        // } else {
                        //  entity->render.visible = 1;
                        // }
                        break;
                    case TYPE_BOWSER:
                        process_bowser_logic(entity, &game);
                        break;
                    case TYPE_FIREBALL:
                        process_fireball_logic(entity, &game);
                        break;
                    default:
                        if (game.camera_pos < 355 || bowser_alive == 0) {
                            entity->position.x -= game.camera_velocity;
                        }

                        if(entity->position.x < game.camera_start) {
                            entity->state.active = 0;
                        }

                        if(entity->position.x > game.camera_start +
CAMERA_SIZE) {
                            entity->render.visible = 0;
                        } else {
                            entity->render.visible = 1;
                        }
                        break;
                }
            }

            animate_entity(&game, entity, frame_counter);
        }
        flush_frame(&game, frame_select);
        frame_select = !frame_select;

        if (frame_counter % 100 == 0)
            printf("camera pos = %d\n", game.camera_pos);
        usleep(16667);
    }

    pthread_cancel(input_thread);
    pthread_join(input_thread, NULL);
    close(vga_ball_fd);
    return 0;
}
```

Game_struct.c

```
#include "game_struct.h"
#include "game_animation.h"
#include <stdlib.h>
#include <math.h>

enum contact hitbox_contact(const Entity *A, const Entity *B) {
    float leftA = A->position.x;
    float rightA = A->position.x + A->position.width;
    float topA = A->position.y;
    float bottomA = A->position.y + A->position.height;

    float leftB = B->position.x;
    float rightB = B->position.x + B->position.width;
    float topB = B->position.y;
    float bottomB = B->position.y + B->position.height;

    // Check for no collision
    if (rightA <= leftB || leftA >= rightB || topA >= bottomB || bottomA
<= topB) {
        return NONE;
    }

    // Calculate Overlap depths
    float leftOverlap = rightA - leftB;
    float rightOverlap = rightB - leftA;
    float topOverlap = bottomA - topB;
    float bottomOverlap = bottomB - topA;

    // Determine the minimal Overlap direction
    float minOverlap = fmin(fmin(leftOverlap, rightOverlap),
fmin(topOverlap, bottomOverlap));

    if (minOverlap == leftOverlap) {
        return RIGHT;
    } else if (minOverlap == rightOverlap) {
        return LEFT;
    } else if (minOverlap == topOverlap) {
        return DOWN;
    } else if (minOverlap == bottomOverlap) {
        return UP;
    }

    return NONE;
}

void new_game(Game *game) {
    game->camera_pos = 70;
    game->camera_start = 70;
    game->game_state = GAME_START;
    game->camera_velocity = 0;

    // Initialize all entities to inactive
    for (int i = 0; i < MAX_ENTITIES; i++) {
        game->entities[i].state.active = 0;
    }

    // Initialize Mario
    Entity *mario = &game->entities[0];
    *mario = (Entity){
        .position = {128, 128, 16, 16},
        .motion = {0, 0, 0, 0},
        .render = {ANI_MARIO_S_NORMAL, 1, 0},
        .state = {1, STATE_NORMAL, 0, TYPE_MARIO_SMALL}
    };

    // Initialize Goomba
    Entity *goomba = &game->entities[1];
    *goomba = (Entity){
        .position = {300, GROUND_LEVEL - 16, 16, 16},
        .motion = {1.0, 0, 0, 0},
        .render = {ANI_GOOMBA_NORMAL, 1, 0},
        .state = {1, STATE_NORMAL, 0, TYPE_GOOMBA}
    };

    // Initialize Ground
    Entity *ground = &game->entities[2];
    *ground = (Entity){
        .position = {70, GROUND_LEVEL, CAMERA_SIZE, 32},
        .motion = {0, 0, 0, 0},
        .render = {0, 1, 0},
        .state = {1, STATE_NORMAL, 0, TYPE_GROUND}
    };

    // Initialize blocks on either side of the Goomba
    Entity *block_left = &game->entities[3];
    *block_left = (Entity){
        .position = {280, GROUND_LEVEL - 16, 16, 16},
        .motion = {0, 0, 0, 0},
        .render = {0, 1, 0},
        .state = {1, BLOCK_NORMAL, 0, TYPE_BLOCK_B_2}
    };

    Entity *block_right = &game->entities[4];
    *block_right = (Entity){
        .position = {400, GROUND_LEVEL - 16, 16, 16},
        .motion = {0, 0, 0, 0},
        .render = {0, 1, 0},
        .state = {1, BLOCK_NORMAL, 0, TYPE_BLOCK_B_2}
    };

    Entity *tube = &game->entities[5];
    *tube = (Entity){
        .position = {200, GROUND_LEVEL - 50, 32, 64},
        .motion = {0,0,0,0},
        .render = {ANI_TUBE_B, 1, 0},
        .state = {1, STATE_NORMAL, 0, TYPE_TUBE}
    };

    // // Initialize Ground
    // Entity *ground2 = &game->entities[6];
    // *ground2 = (Entity){
    //   .position = {CAMERA_SIZE + GROUND_PIT_WIDTH + 70, GROUND_LEVEL,
CAMERA_SIZE, 32},
    //   .motion = {0, 0, 0, 0},
    //   .render = {0, 0, 0},
    //   .state = {1, STATE_NORMAL, 0, TYPE_GROUND}
    // };

    Entity *block_left2 = &game->entities[6];
    *block_left2 = (Entity){
        .position = {CAMERA_SIZE + 70, GROUND_LEVEL-16, 16, 16},
        .motion = {0, 0, 0, 0},
        .render = {0, 1, 0},
        .state = {1, BLOCK_NORMAL, 0, TYPE_BLOCK_B_3}
    };

    Entity *bowser = &game->entities[7];
    *bowser = (Entity) {
        .position = {CAMERA_SIZE + 70 + 100, 128, 32, 32},
        .motion = {0, 0, 0, 0},
        .render = {ANI_BOWSER_NORMAL, 1, 0},
        .state = {1, STATE_NORMAL, 0, TYPE_BOWSER}
    };

    Entity *block_right2 = &game->entities[8];
    *block_right2 = (Entity){
        .position = {CAMERA_SIZE + 70 + 400,GROUND_LEVEL-16, 16, 16},
        .motion = {0, 0, 0, 0},
        .render = {0, 1, 0},
        .state = {1, BLOCK_NORMAL, 0, TYPE_BLOCK_B_3}
    };

    Entity *peach = &game->entities[9];
    *peach = (Entity) {
        .position = {CAMERA_SIZE + 70 + 600, GROUND_LEVEL - 40,  32, 40},
        .motion = {0, 0, 0, 0},
        .render = {ANI_PEACH_NORMAL, 1, 0},
        .state = {1, STATE_NORMAL, 0, TYPE_PEACH}
    };

    Entity *fireball = &game->entities[10];
    *fireball = (Entity) {
        .position = {200, GROUND_LEVEL - 48, 14, 15},
        .motion = {0, 0, 0, 0},
        .render = {ANI_FIREBALL_NORMAL, 1, 0},
        .state = {1, STATE_NORMAL, 0, TYPE_FIREBALL}
    };
}
```

Game_animation.h

```
#ifndef _GAME_ANIMATION_H
#define _GAME_ANIMATION_H

#include "game_struct.h"

// Frame Limit
#define FRAME_LIMIT 6000

// Mario Animation
#define ANI_MARIO_S_NORMAL 0
#define ANI_MARIO_S_WALK1 1
#define ANI_MARIO_S_WALK2 2
#define ANI_MARIO_S_WALK3 3
#define ANI_MARIO_S_SHUT 4
#define ANI_MARIO_S_JUMP 5
#define ANI_MARIO_S_DEAD 6
#define ANI_MARIO_L_NORMAL 7
#define ANI_MARIO_L_WALK1 8
#define ANI_MARIO_L_WALK2 9
#define ANI_MARIO_L_WALK3 10
#define ANI_MARIO_L_SHUT 11
#define ANI_MARIO_L_JUMP 12
#define ANI_MARIO_L_SIT 13
#define ANI_MARIO_M_NORMAL 14
#define ANI_MARIO_L_HIT 15
#define ANI_MARIO_S_HIT 16
#define ANI_MARIO_S_HANG 17
#define ANI_MARIO_L_HANG 18

// Goomba Animation
#define ANI_GOOMBA_NORMAL 0
#define ANI_GOOMBA_HIT 1

// Tube Animation
#define ANI_TUBE_H 0
#define ANI_TUBE_B 1

// Block Animation
#define ANI_BLOCK_ITEM1 0
```

```c
#define ANI_BLOCK_ITEM2 1
#define ANI_BLOCK_ITEM3 2
#define ANI_BLOCK_ITEM_HIT 3
#define ANI_BLOCK_ITEM_EMP 4
#define ANI_BLOCK_A1 5
#define ANI_BLOCK_A2 6
#define ANI_BLOCK_A_HIT 7
#define ANI_BLOCK_B 8
#define ANI_BLOCK_A_H2 9
#define ANI_BLOCK_A_H3 10
#define ANI_BLOCK_A_H8 11
#define ANI_BLOCK_B_V2 12
#define ANI_BLOCK_B_V3 13
#define ANI_BLOCK_B_V4 14
#define ANI_BLOCK_B_16 15

#define ANI_BOWSER_NORMAL 0

#define ANI_FIREBALL_NORMAL 0

#define ANI_PEACH_NORMAL 0

// Function pointer type for entity animation functions
typedef void (*AnimateFunc)(Game *game, Entity *entity, int f_counter);

// Animation function declarations
void animate_mario(Game *game, Entity *entity, int f_counter);
void animate_goomba(Game *game, Entity *entity, int f_counter);
void animate_block(Game *game, Entity *entity, int f_counter);
void animate_tube(Game *game, Entity *entity, int f_counter);
void animate_bowser(Game *game, Entity *entity, int f_counter);
void animate_fireball(Game *game, Entity *entity, int f_counter);
void animate_peach(Game *game, Entity *entity, int f_counter);
// Generic function to animate any entity
void animate_entity(Game *game, Entity *entity, int f_counter);

#endif // _GAME_ANIMATION_H
```

Game_animation.c

```c
#include "game_animation.h"
```

```
#include <stdio.h>
#include <math.h>

void animate_entity(Game *game, Entity *entity, int f_counter) {
    if (entity->state.active && entity->render.visible) {
        switch (entity->state.type) {
            case TYPE_MARIO_SMALL:
            case TYPE_MARIO_LARGE:
                animate_mario(game, entity, f_counter);
                break;
            case TYPE_GOOMBA:
                animate_goomba(game, entity, f_counter);
                break;
            case TYPE_BLOCK_A:
            case TYPE_BLOCK_B_1:
            case TYPE_BLOCK_B_2:
            case TYPE_BLOCK_B_3:
            case TYPE_BLOCK_B_4:
            case TYPE_BLOCK_B_16:
            case TYPE_BLOCK_A_H_8:
            case TYPE_BLOCK_OBJ_C:
            case TYPE_BLOCK_OBJ_M:
                animate_block(game, entity, f_counter);
                break;
            case TYPE_TUBE:
                animate_tube(game, entity, f_counter);
                break;
            case TYPE_GROUND:
                break;
            case TYPE_BOWSER:
                animate_bowser(game, entity, f_counter);
                break;
            case TYPE_PEACH:
                animate_peach(game, entity, f_counter);
                break;
            case TYPE_FIREBALL:
                animate_fireball(game, entity, f_counter);
                break;
            default:
                break;
        }
    }
}

void animate_mario(Game *game, Entity *entity, int f_counter) {
    if (entity->state.active && entity->render.visible) {
        const int frame_count = 5; // Update sprite every 10 frames
        int rel_counter = f_counter % frame_count; // Calculate frame
counter relative to the frame_count

        float dead_v = -4.7f;
        float dead_acc = 0.16f;

        if (rel_counter == 0) { // Only update sprite state every 10
frames
            switch (entity->state.state) {
                case STATE_NORMAL:
                    if (entity->motion.vx == 0) {
                        entity->render.pattern_code = ANI_MARIO_S_NORMAL;
                    // } else if (entity->motion.vx * entity->motion.ax <
0) {
                        // entity->render.pattern_code = ANI_MARIO_S_SHUT;
                    } else if (fabs(entity->motion.vx) > 2*MOTION_MIN) {
                        int ani_stage = (f_counter / 6) % 3; // Change to
walk animation frames
                        entity->render.pattern_code = (ani_stage == 0) ?
ANI_MARIO_S_WALK1 :
                                                     (ani_stage == 1) ?
ANI_MARIO_S_WALK2 :
                                                     ANI_MARIO_S_WALK3;
                    }
                    break;
            case STATE_HIT:
                entity->render.pattern_code = (rel_counter / 3) % 2 ?
ANI_MARIO_L_HIT : ANI_MARIO_S_HIT;
                if (rel_counter == 0) entity->position.y += 3;
                if (rel_counter > 20) entity->state.state = STATE_NORMAL;
                break;
            case STATE_DEAD:
                /*Doesn't really animate the death as STATE_DEAD is caught
by game loop*/
                entity->render.pattern_code = ANI_MARIO_S_DEAD;
                if (rel_counter == 0) entity->motion.vy = dead_v;
                else if (rel_counter > 30) {
                    entity->position.y += entity->motion.vy;
                    entity->motion.vy += dead_acc;
                    if (entity->position.y > 1.01*GROUND_LEVEL)
entity->state.state = STATE_DEAD;
                }
                break;
            case STATE_LARGE:
                if (entity->motion.ax == 0) {
                    entity->render.pattern_code = ANI_MARIO_L_NORMAL;
                } else if (entity->motion.ax * entity->motion.vx < 0) {
                    entity->render.pattern_code = ANI_MARIO_L_SHUT;
                } else {
                    int ani_div = (rel_counter / 6) % 3;
                    entity->render.pattern_code = (ani_div == 0) ?
ANI_MARIO_L_WALK1 :
                                                  (ani_div == 1) ?
ANI_MARIO_L_WALK2 :
                                                  ANI_MARIO_L_WALK3;
                }
                break;
        }

        entity->render.visible = (entity->state.state == STATE_NORMAL ||
entity->state.state == STATE_LARGE) ? 1 : (rel_counter / 15) % 2;
        }
    }
}

void animate_goomba(Game *game, Entity *entity, int f_counter) {
    if (entity->state.active) {
        int frame_count = FRAME_LIMIT;

        int rel_counter = (f_counter - entity->state.animate_frame_counter
+ frame_count) % frame_count;
```

```
        if (entity->state.state == STATE_NORMAL) {
            entity->render.pattern_code = ANI_GOOMBA_NORMAL;
        }
        entity->render.visible = 1;
    }
}

void animate_bowser(Game *game, Entity *entity, int f_counter) {
    if (entity->state.active) {
        int frame_count = FRAME_LIMIT;

        int rel_counter = (f_counter - entity->state.animate_frame_counter
+ frame_count) % frame_count;

        if (entity->state.state == STATE_NORMAL) {
            entity->render.pattern_code = ANI_BOWSER_NORMAL;
        }
        entity->render.visible = 1;
    }
}

void animate_peach(Game *game, Entity *entity, int f_counter) {
    if (entity->state.active) {
        int frame_count = FRAME_LIMIT;

        int rel_counter = (f_counter - entity->state.animate_frame_counter
+ frame_count) % frame_count;

        if (entity->state.state == STATE_NORMAL) {
            entity->render.pattern_code = ANI_PEACH_NORMAL;
        }
        entity->render.visible = 1;
    }
}

void animate_fireball(Game *game, Entity *entity, int f_counter) {
    if (entity->state.active) {
        int frame_count = FRAME_LIMIT;
        int rel_counter = (f_counter - entity->state.animate_frame_counter
+ frame_count) % frame_count;

        if (entity->state.state == STATE_NORMAL) {
            entity->render.pattern_code = ANI_PEACH_NORMAL;
        }
        entity->render.visible = 1;
    }
}

void animate_tube(Game *game, Entity *entity, int f_counter) {
    if (entity->state.active) {
        entity->render.visible = 1;
    }
}

void animate_block(Game *game, Entity *entity, int f_counter) {
    if (entity->state.active) {
        int frame_count = FRAME_LIMIT;
        int counter = (f_counter >= entity->state.animate_frame_counter) ?
                      f_counter : f_counter + frame_count;
        int rel_counter = counter - entity->state.animate_frame_counter;

        switch (entity->state.state) {
            case STATE_NORMAL:
                switch (entity->state.type) {
                    case TYPE_BLOCK_A:
                        entity->render.pattern_code = ANI_BLOCK_A1;
                        break;
                    case TYPE_BLOCK_B_1:
                        entity->render.pattern_code = ANI_BLOCK_B;
                        break;
                    case TYPE_BLOCK_B_2:
                        entity->render.pattern_code = ANI_BLOCK_B_V2;
                        break;
                    case TYPE_BLOCK_B_3:
                        entity->render.pattern_code = ANI_BLOCK_B_V3;
                        break;
                    case TYPE_BLOCK_B_4:
                        entity->render.pattern_code = ANI_BLOCK_B_V4;
                        break;
                    case TYPE_BLOCK_B_16:
                        entity->render.pattern_code = ANI_BLOCK_B_16;
                        break;
                    case TYPE_BLOCK_A_H_8:
                        entity->render.pattern_code = ANI_BLOCK_A_H8;
                        break;
                    case TYPE_BLOCK_OBJ_C:
                    case TYPE_BLOCK_OBJ_M:
                        entity->render.pattern_code = ANI_BLOCK_ITEM1;
                        break;
                    default:
                        entity->render.pattern_code = ANI_BLOCK_ITEM_EMP;
                        break;
                }
                break;
            default:
                entity->render.pattern_code = ANI_BLOCK_ITEM_EMP;
                break;
        }
    }
}
```

Game_struct.h

```
#ifndef _MARIO_GAME_STRUCT
#define _MARIO_GAME_STRUCT

#include <stdint.h>

#define MAX_ENTITIES 128

#define GRAVITY 0.23f
#define MAX_SPEED_H 1.85f
#define MAX_SPEED_V 4.6f
#define LOAD_LIMIT (5*16)
#define CAMERA_SIZE 558
#define GROUND_LEVEL 368

#define GROUND_PIT_WIDTH 75
```

```
#define WALK_ACC (0.09)
#define SHUT_ACC (0.12)
#define JUMP_INIT_V_SMALL (4.6)
#define JUMP_INIT_V_LARGE (5.6)
#define MAX_SPEED_V_JUMP (8.1)
#define FRICTION (0.08)
#define MOTION_MIN 0.01f

enum EntityState {
    STATE_NORMAL,
    STATE_ANIMATE,
    STATE_HIT,
    STATE_DEAD,
    STATE_LARGE,
    BLOCK_NORMAL,
    BLOCK_ANIMATE
};

enum contact {
    NONE, LEFT, RIGHT, UP, DOWN
};

enum EntityType {
    TYPE_MARIO_SMALL,
    TYPE_MARIO_LARGE,
    TYPE_GOOMBA,
    TYPE_PEACH,
    TYPE_FIREBALL,
    TYPE_BLOCK_A,
    TYPE_BLOCK_B_1,
    TYPE_BLOCK_B_2,
    TYPE_BLOCK_B_3,
    TYPE_BLOCK_B_4,
    TYPE_BLOCK_B_16,
    TYPE_BLOCK_A_H_8,
    TYPE_BLOCK_OBJ_C,
    TYPE_BLOCK_OBJ_M,
    TYPE_TUBE,
    TYPE_GROUND,
    TYPE_BOWSER,
    TYPE_EMP
};

typedef struct {
    float x, y;
    int width, height;
} PositionComponent;

typedef struct {
    float vx, vy;
    float ax, ay;
} MotionComponent;

typedef struct {
    uint32_t pattern_code;
    int visible;
    int flip;
} RenderComponent;

typedef struct {
    int active;
    int state;
    int animate_frame_counter;
    int type;
} StateComponent;

typedef struct {
    PositionComponent position;
    MotionComponent motion;
    RenderComponent render;
    StateComponent state;
} Entity;


typedef struct {
    int camera_start;
    int camera_pos;
    float camera_velocity;
    int game_state;
    Entity entities[MAX_ENTITIES];
```

```
} Game;

enum {
    GAME_START,
    GAME_NORMAL,
    GAME_END
};

void new_game(Game *game);
enum contact hitbox_contact(const Entity *A, const Entity *B);

#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
 *
http://www.dreamincode.net/forums/topic/148707-introduction-to-using-libus
b-10/
 * http://www.usb.org/developers/devclass_docs/HID1_11.pdf
 * http://www.usb.org/developers/devclass_docs/Hut1_11.pdf
 */

/*
 * Find and return a USB keyboard device or NULL if not found
 * The argument con
 *
 */
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;
    /* Start the library */
    if ( libusb_init(NULL) < 0 ) {
        fprintf(stderr, "Error: libusb_init failed\n");
        exit(1);
    }

    /* Enumerate all the attached USB devices */
    if ( (num_devs = libusb_get_device_list(NULL, &devs)) < 0 ) {
        fprintf(stderr, "Error: libusb_get_device_list failed\n");
        exit(1);
    }

    /* Look at each device, remembering the first HID device that speaks
          the keyboard protocol */

    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
```

Vga_ball

```
/* * Device driver for the VGA video generator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod vga_ball.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree vga_ball.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"

/* Device registers */
#define BG_RED(x) (x)
#define BG_GREEN(x) ((x)+1)
#define BG_BLUE(x) ((x)+2)
#define BG_BALL_XL(x) ((x)+3)
#define BG_BALL_XH(x) ((x)+4)
#define BG_BALL_YL(x) ((x)+5)
#define BG_BALL_YH(x) ((x)+6)

/*
 * Information about our device
 */
struct vga_ball_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory
*/
        vga_ball_color_t background;
    vga_ball_coordinate_t coordinate;
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
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(vga_ball_coordinate_t *coordinate)
{
    iowrite8(coordinate->xl, BG_BALL_XL(dev.virtbase) );
    iowrite8(coordinate->xh, BG_BALL_XH(dev.virtbase) );
    iowrite8(coordinate->yl, BG_BALL_YL(dev.virtbase) );
    iowrite8(coordinate->yh, BG_BALL_YH(dev.virtbase) );
    dev.coordinate = *coordinate;
}

static void write_hw(int addr, int info)
{
    iowrite32(info, dev.virtbase + addr);
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
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_hw(vla.addr, vla.info);
        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;

    default:
        return -EINVAL;
    }

    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations vga_ball_fops = {
    .owner      = THIS_MODULE,
    .unlocked_ioctl = vga_ball_ioctl,
};

/* Information about our device for the "misc" framework -- like a char
dev */
static struct miscdevice vga_ball_misc_device = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = DRIVER_NAME,
    .fops       = &vga_ball_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init vga_ball_probe(struct platform_device *pdev)
{
        vga_ball_color_t beige = { 0x0D, 0x0D, 0x0D };
    int ret;

    /* Register ourselves as a misc device: creates /dev/vga_ball */
    ret = misc_register(&vga_ball_misc_device);

    /* Get the address of our registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret) {
        ret = -ENOENT;
        goto out_deregister;
    }

    /* Make sure we can use these registers */
    if (request_mem_region(dev.res.start, resource_size(&dev.res),
                   DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

    /* Arrange access to our registers */
    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_mem_region;
    }

    /* Set an initial color */
        write_background(&beige);

    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;
}

/* Clean-up code: release resources */
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;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#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

/* Information for registering ourselves as a "platform" driver */
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),
};

/* Called when the module is loaded: set things up */
static int __init vga_ball_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_probe(&vga_ball_driver, vga_ball_probe);
}

/* Calball when the module is unloaded: release resources */
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_ball.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 char xl, xh, yl, yh;
} vga_ball_coordinate_t;

typedef struct {
  vga_ball_color_t background;
    vga_ball_coordinate_t coordinate;
    int addr;
    int info;
} 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
*)

#endif
```

Configure

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>

#define WM8731_ADDR 0x1a  // Should be fine

void write_register(int file, int reg, int value) {
    unsigned char buffer[2];
    buffer[0] = (reg << 1) | ((value >> 8) & 0x01);  // Register address
and MSB of value
    buffer[1] = value & 0xFF;  // LSB of value
    if (write(file, buffer, 2) != 2) {
        perror("Failed to write to the i2c bus");
    }
}

int main() {
    int file;
    char *filename = "/dev/i2c-1";  // Verify which I2C bus the codec is
connected to

    // Open I2C bus
    if ((file = open(filename, O_RDWR)) < 0) {
        perror("Failed to open the I2C bus");
        return -1;
    }

    // Configure the I2C client
    if (ioctl(file, I2C_SLAVE, WM8731_ADDR) < 0) {
        perror("Failed to acquire bus access and/or talk to slave");
        close(file);
        return -1;
    }

    // Initialize WM8731
    write_register(file, 0x0F, 0x000);  // Reset device
    write_register(file, 0x07, 0x001);  // Enable DAC
    write_register(file, 0x08, 0x015);  // Enable output
    write_register(file, 0x0A, 0x000);  // Disable mute
    write_register(file, 0x02, 0x017);  // Set headphone volume (check
range)
    write_register(file, 0x0E, 0x021);  // Enable DAC and set audio format
(16-bit I2S)
    write_register(file, 0x12, 0x001);  // Enable Clock

    close(file);
    return 0;
}
//RUN: arm-linux-gnueabihf-gcc -o configure_wm8731 configure_wm8731.c
```

toBytes.py

```
from PIL import Image
import math
```

```python
def find_closest_color(rgb, palette):
    min_distance = float('inf')
    index_selected = 0
    for index, color in enumerate(palette):
        distance = math.sqrt(sum((c1 - c2) ** 2 for c1, c2 in zip(rgb,
color)))
        if distance < min_distance:
            min_distance = distance
            index_selected = index
    return index_selected

file_name = input("Enter File Name: ")
im = Image.open(file_name)
pix = im.load()

print("Image Size: ", im.size)
print("Image Mode: ", im.mode)

Color_Plate = [
    (255, 0, 0),
    (255, 204, 102),
    (255, 255, 255),
    (20, 20, 20),
]

byte_stream = []

for i in range(im.size[1]):  # y
    for j in range(im.size[0]):  # x
        rgb = pix[j, i][:3]  # Get RGB (ignore alpha if present)
        if rgb not in Color_Plate:
            index = find_closest_color(rgb, Color_Plate)
        else:
            index = Color_Plate.index(rgb)
        byte_stream.append(index)

print("Color Plate Info:")
for i, color in enumerate(Color_Plate):
    print(f"{i}: {color}")
print("Coded File Length (Bytes): ", len(byte_stream))

saved_file_name = input("Enter Saved Filename: ")
with open(saved_file_name, 'w') as f:
    write_stream = '\n'.join(f'{a:01x}' for a in byte_stream)
    f.write(write_stream)
```