CSEE4840 Embedded Systems Final Report:

GridBrawl Game

Siyao Yu(sy3342)

Sitao Zhang(sz3419)

Anyongyong Zhao(az2932)

Spring 2026

Introduction

Our team designed a two-player, tile-based competitive game on the DE1-SoC FPGA platform. The game combines real-time player movement, territory capture, bomb placement, destructible walls, VGA graphics, and audio feedback. The HPS handles the main game logic and player input, while the FPGA is responsible for hardware-accelerated graphics and audio output.
The game takes place on a closed 20 × 15 tile map displayed on a 640 × 480 VGA screen. The map contains hard walls, soft walls, and capturable tiles. Hard walls are permanent obstacles, while soft walls can be destroyed by bomb explosions to create new paths. Two players compete by moving across the map and capturing tiles. Players can place bombs to destroy soft walls or attack the opponent. If a player is hit by an explosion, that player dies, and all tiles previously captured by that player are reset to neutral blue tiles. The first player to capture 50 tiles wins the game. This project demonstrates hardware/software co-design on the DE1-SoC. The software controls gameplay rules and state updates, while the FPGA provides real-time VGA rendering and sound playback for a responsive game experience.

1. System Block Diagram

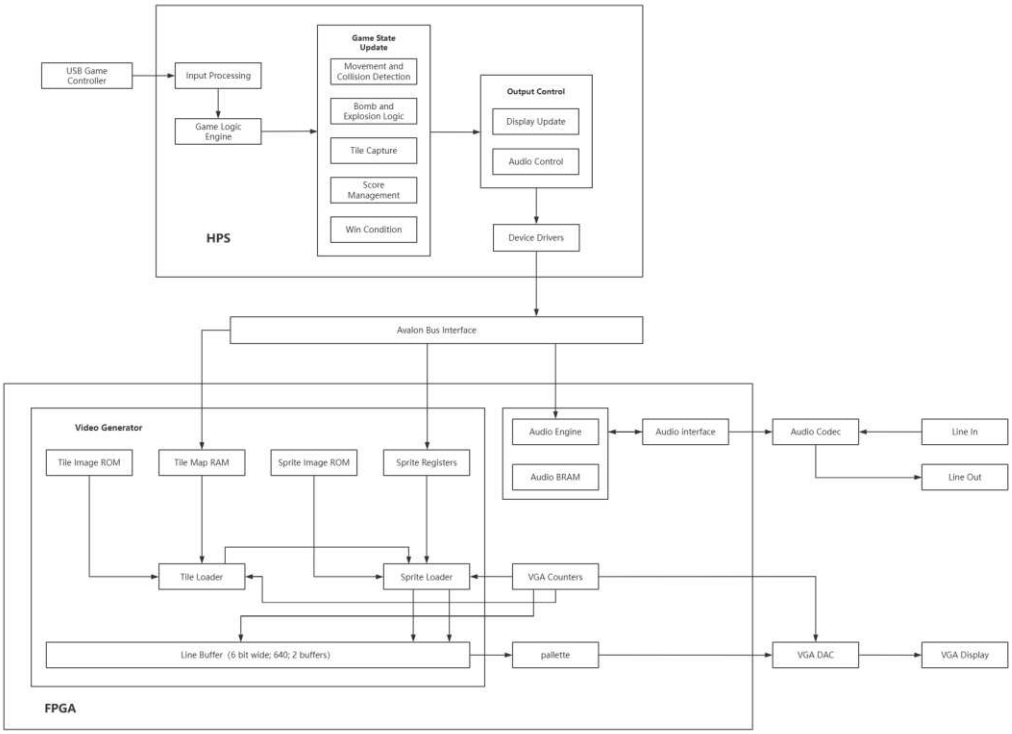

Figure 1. Hardware/Software System Block Diagram for GridBrawl on the DE1-SoC

The system is divided between software control on the HPS and real-time output generation on the FPGA fabric. The HPS runs the game logic, processes USB controller input, updates player movement, collision state, scoring, bombs, and sound events. The FPGA implements the timing-critical video and audio hardware, including scanline-based VGA rendering and deterministic audio playback. Communication between the HPS and FPGA is performed through Avalon-MM memory-mapped interfaces. The HPS writes tile IDs into the Tile Map RAM, updates Sprite Registers with sprite position and enable information, and writes sound event IDs to the FPGA audio controller. These memory-mapped hardware blocks provide the current game state to the rendering and audio pipelines without requiring the FPGA to execute high-level game rules.

The VGA subsystem uses a scanline-based rendering pipeline. For each line, the tile loader reads the Tile Map RAM and Tile Image ROM to generate background pixels. The sprite loader then reads the Sprite Registers and Sprite Image ROMs to overlay dynamic objects such as players, bombs, explosions, and UI elements. The line buffer stores the generated scanline, and the palette module converts stored color indices into RGB values for VGA output. The audio subsystem runs in parallel with the video pipeline. The FPGA audio controller selects background music or event-triggered sound effects from the audio ROM and sends sample data through the Intel audio interface to the WM8731 audio CODEC. This hardware/software partition allows asynchronous game logic execution on the HPS while the FPGA continuously maintains deterministic real-time video and audio timing.

2. Hardware design

2.1 Graphics Display

The graphics display subsystem is implemented on the FPGA as a hardware VGA renderer. The design outputs a 640 × 480 VGA image using a tile-based background layer and a sprite overlay layer. This structure matches the 20 × 15 game map, where each tile is 32 × 32 pixels. The FPGA generates pixels in real time by expanding compact tile IDs and sprite states into palette-indexed VGA output.

VGA Control: Tile Drawing

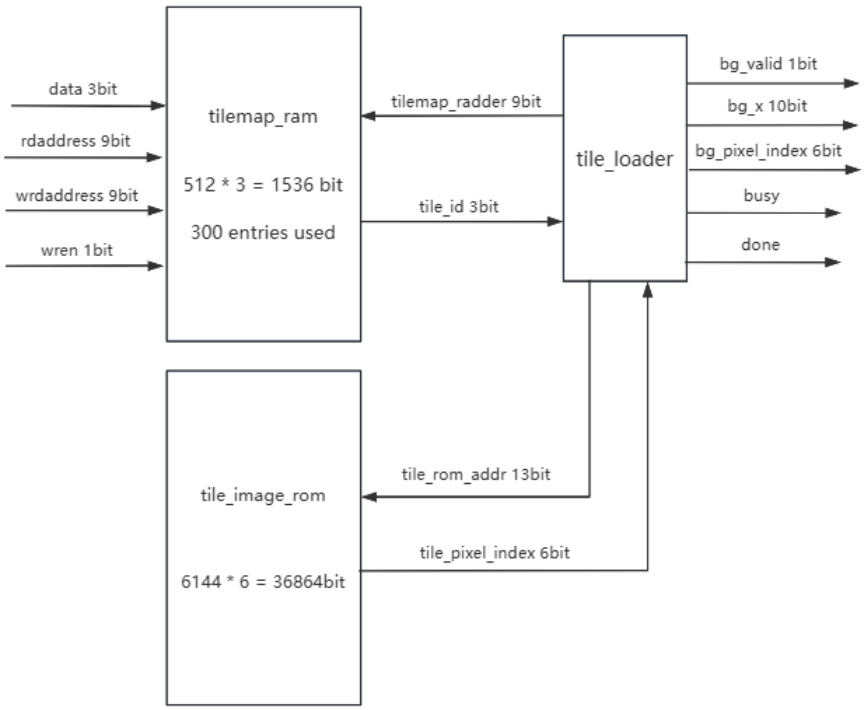

Figure 2. VGA Control: Tile Drawing

The tile drawing pipeline generates the background layer of the game screen. As shown in Figure 2, the Tile Map RAM stores the logical map of the game. Although the physical RAM has 512 entries, only 300 entries are used, corresponding to the 20 × 15 game board. Each entry stores a 3-bit tile ID. The tile ID represents the type of background block, such as empty floor, colored tile, hard wall, or soft wall.

During VGA rendering, the tile_loader reads from the RAM using a 9-bit tilemap_raddr. The returned tile_id is then used to address the Tile Image ROM.

The Tile Image ROM stores the actual pixel pattern of each tile. Each tile is 32 × 32 pixels, and each pixel is represented by a 6-bit palette index. The tile_loader calculates the Tile Image ROM address using three pieces of information: the tile ID, the row offset within the tile, and the column offset within the tile. In the implementation, the address is formed from:

tile_rom_addr = {tile_id, row_in_tile, pixel_column}

The tile loader then streams one scanline of background pixels. For every generated pixel, it outputs:

- bg_valid
- bg_x

- bg_pixel_index

bg_valid indicates that the current background pixel is valid, bg_x gives the horizontal screen coordinate, and bg_pixel_index gives the 6-bit color index. The busy and done signals are used by the top-level rendering controller to coordinate scanline rendering.

This tile pipeline is efficient because only tile IDs are stored in RAM, while repeated pixel artwork is stored once in ROM. Therefore, the game can update the map using small memory writes while the FPGA expands those tile IDs into full VGA pixels in hardware.

VGA Control: Sprite Drawing

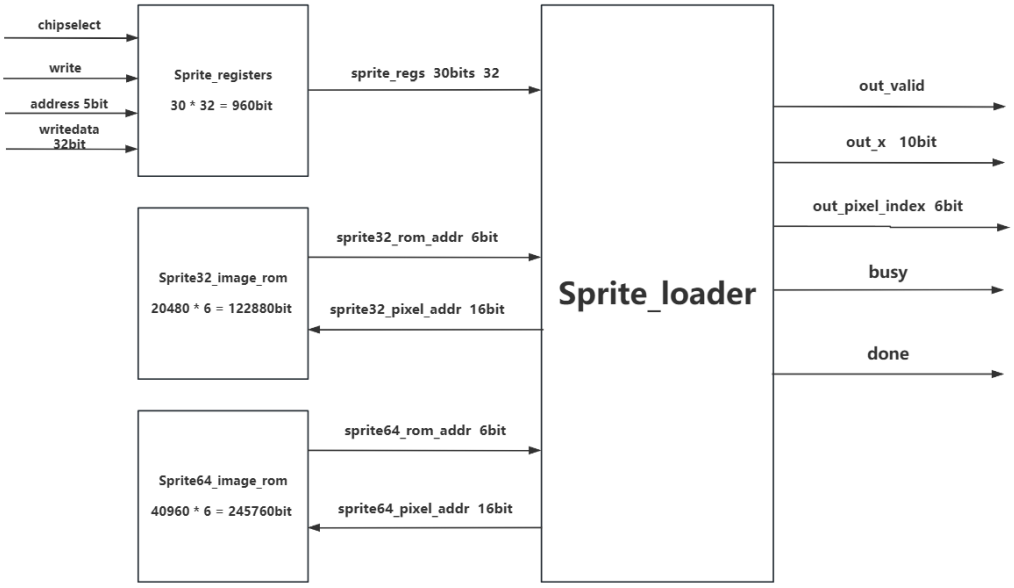

Figure 3. VGA Control: Sprite Drawing

The sprite drawing pipeline is responsible for dynamic objects, including players, bombs, explosions, and UI elements. As shown in Figure 3, sprite states are stored in Sprite Registers. The design contains 30 sprite registers, each 32 bits wide, for a total of 960 bits of sprite state storage. Each sprite register stores the enable bit, sprite ID, and screen position.

The sprite loader receives all 30 sprite registers and checks them one by one for the current scanline. If a sprite is enabled and overlaps the current vcount, the sprite loader calculates the corresponding row inside the sprite image and fetches pixel data from the sprite ROM,

including Sprite32 Image ROM: stores 32 × 32 sprites and Sprite64 Image ROM: stores 64 × 64 sprites.

The 32 × 32 ROM is used for most gameplay objects such as players, bombs, and explosion tiles. The 64 × 64 ROM is used for larger UI graphics, such as player labels and win text. Each sprite pixel is also represented as a 6-bit palette index.

The sprite loader outputs:

- out_valid
- out_x
- out_pixel_index
- busy
- done

Only non-transparent pixels are written to the line buffer. In this design, palette index 0 is treated as transparent. This allows sprites to be drawn over the tile background without covering the entire rectangular sprite area. Later sprite writes can overwrite earlier pixels at the same x coordinate, which creates the intended layering effect.

VGA Top Module and Line Buffer

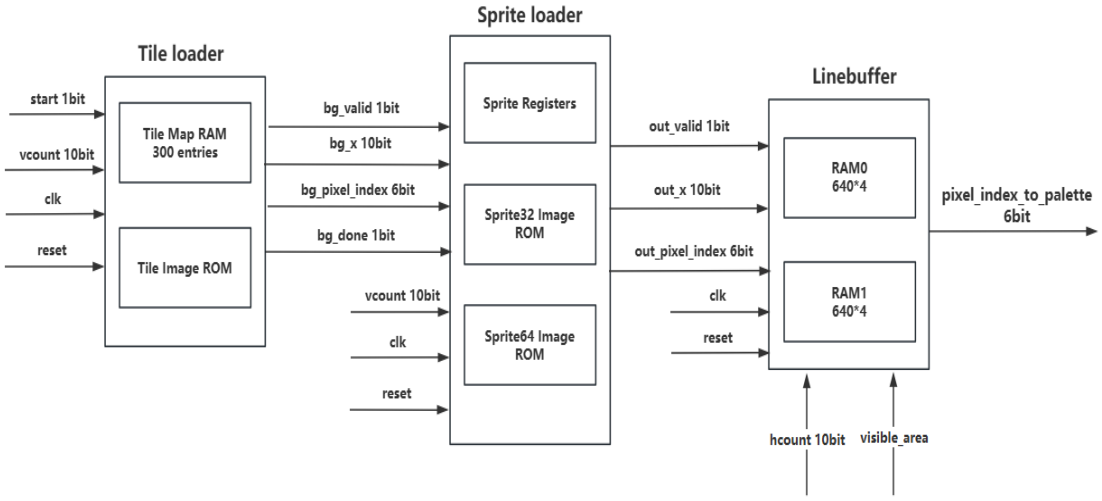

Figure 4. VGA Top Module and Line Buffer

The VGA top module connects the tile loader, sprite loader, line buffer, palette, and VGA timing logic into a complete rendering pipeline. As shown in Figure 4, the rendering process is performed one scanline at a time.

For each visible scanline, the tile loader first generates the background pixels. These pixels are passed to the sprite loader through:

- bg_valid
- bg_x
- bg_pixel_index
- bg_done

After the background line is complete, the sprite loader overlays all active sprites that intersect the same scanline. The final pixel stream is then sent to the line buffer using:

- out_valid
- out_x
- out_pixel_index

The line buffer is implemented as a double buffer. It contains two 640-pixel buffers, shown as RAM0 and RAM1 in Figure 4. While the VGA output reads from the front buffer, the renderer writes the next scanline into the back buffer. At a safe scanline boundary, the front and back buffers are swapped.

This double-buffered line design prevents visible tearing. The system never swaps buffers in the middle of a displayed scanline. Instead, the rendering controller waits until the current line reaches its end before changing the active buffer. This is important because VGA output is continuous and cannot pause while the FPGA finishes rendering.

The line buffer outputs a 6-bit pixel_index_to_palette. This value is sent to the palette module, which converts the 6-bit index into a 24-bit RGB value. The final RGB output is connected to the VGA DAC through:

- VGA_R
- VGA_G
- VGA_B

The VGA timing logic generates horizontal and vertical counters, synchronization signals, blanking, and the VGA clock. The system uses a 50 MHz clock, where each visible VGA pixel corresponds to two 50 MHz cycles. The actual x coordinate is derived from the horizontal counter, while the y coordinate is given by the vertical counter.

Overall, the VGA top module implements a hardware rendering pipeline similar to a simple 2D graphics engine: tile background first, sprite overlay second, palette conversion last, and continuous VGA output in real time.

Graphics Asset Preprocessing Pipeline

Graphics asset preprocessing: The game artwork is converted into palette-indexed image data before synthesis. Instead of storing full 24-bit RGB values for each pixel, every pixel is represented using a compact 6-bit palette index. This significantly reduces the memory required for Tile ROM and Sprite ROM storage.

ROM initialization: The processed tile and sprite image data are stored in MIF files that initialize the FPGA ROM blocks. These ROMs are accessed by the tile loader and sprite loader during real-time VGA rendering.

Transparency handling: Transparent sprite pixels are encoded using palette index 0. During sprite rendering, pixels with this index are ignored, allowing the existing background pixels in the line buffer to remain visible.

Memory optimization: By preprocessing graphics into fixed-size palette-indexed ROM data before synthesis, the FPGA can directly access compact image assets during rendering without requiring external image decoding or full-frame RGB storage.

2.2 Audio Subsystem

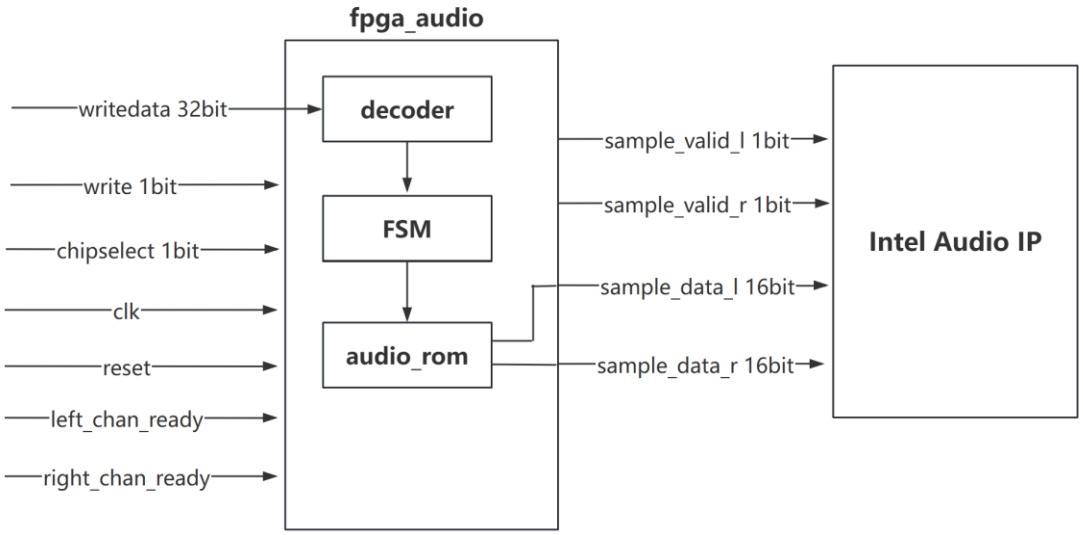

Figure 5. Audio Subsystem

The audio subsystem is implemented as a custom FPGA audio controller connected to the Intel Audio IP and the WM8731 audio CODEC.

Figure 6. Platform Designer integration of the audio IP core

The FPGA then decodes the sound ID and plays the corresponding audio sample from ROM.

As shown in Figure 5, the fpga_audio module contains three main parts:

- decoder
- FSM
- audio_rom

The lower bits of writedata represent the sound ID. When a valid sound effect is written, the decoder selects the corresponding address range in Audio ROM. The FSM then controls

playback. The audio controller normally plays background music in a loop. When a sound effect is triggered, the FSM temporarily switches from background music mode to sound-effect mode, plays the selected sound effect, and then returns to the previous background music position.

The Audio ROM stores 8-bit mono PCM samples. Different address ranges are assigned to different sounds. For example, one range stores background music, while later ranges store explosion, death, victory, and wall break effects. The FPGA reads one 8-bit sample from ROM and converts it into a 16-bit audio sample for both left and right channels.

The output signals to the Intel Audio IP are:

- sample_valid_l
- sample_valid_r
- sample_data_l
- sample_data_r

The Intel Audio IP provides left_chan_ready and right_chan_ready signals. The FPGA audio controller only submits valid samples when both channels are ready. Since the stored PCM data uses an 8 kHz sample rate and the audio IP requests samples at a higher rate, the controller uses a 6-cycle sample-hold mechanism. Each ROM sample is held for six audio-ready cycles before the ROM address advances. This preserves the intended playback speed while still satisfying the ready/valid interface of the audio hardware.

3. Software Design

3.1 Game Logic

- **Overview**

GridBrawl is a two-player, tile-based competitive game running on a 640 × 480 VGA display. The map is divided into a 20 × 15 grid, where each tile is 32 × 32 pixels. Player 1 controls the yellow territory, and Player 2 controls the pink territory. The objective is to capture tiles by moving across the map, while using bombs to destroy soft walls and attack the opponent. The first player whose score reaches 50 captured tiles wins the game.

The game logic is handled on the HPS software side. The FPGA hardware does not make gameplay decisions such as movement, collision, scoring, death, or victory. Instead, the software updates a game_state_t structure every frame, including the tile map, player states, bombs, explosions, scores, and pending sound events. The VGA and audio interface modules then use this state to update the FPGA tile map, sprite registers, and audio controller.

- **Map and Tile Rules**

The map contains six tile types: empty, blue, pink, yellow, hard wall, and soft wall. Blue, yellow, and pink tiles are enterable. Empty tiles, hard walls, and soft walls are not enterable. Empty tiles are mainly used for the UI area, so the UI region naturally blocks player movement without requiring an extra collision box.

Blue tiles are neutral tiles. When a player reaches a blue tile, it changes to that player's color, and the player's score increases by one. If a player reaches the opponent's tile, the tile changes ownership. The current player gains one point, and the opponent loses one point if possible. Hard walls are permanent obstacles and cannot be destroyed. Soft walls are obstacles that can be destroyed by bomb explosions and later become neutral blue tiles.

At reset, the initial map is restored. Player 1 starts at row 1, column 1, and Player 2 starts at row 1, column 18. Both spawn tiles are immediately colored as the corresponding player's territory, so both players start with a score of 1.

- **Player Movement and Collision**

Player movement combines tile-based logic with smooth pixel-level motion. Each player stores a logical tile position and a pixel position. The game runs at 32 ticks per second, and each player moves 2 pixels per tick. Since each tile is 32 pixels wide, moving across one tile takes 16 ticks, or about 0.5 seconds.

A player can start moving only when alive, not already moving, and when the target tile is valid. Once movement begins, the player continues until reaching the target tile. The player cannot turn

in the middle of a tile transition, but the latest input direction is saved as a pending direction, allowing continuous movement after reaching the next tile.

Collision detection prevents players from entering invalid or occupied tiles. A target tile is rejected if it is outside the map, not enterable, occupied by an active bomb, occupied by the other player, or reserved as the other player's movement target. If both players try to move into the same valid tile in the same frame, both movement requests are canceled. This avoids overlap and gives no priority advantage to either player.

Tile capture happens only after a player reaches a tile, not when movement starts. This keeps scoring synchronized with the player's final logical position.

- **Bombs and Explosions**

Each player has one bomb slot and one corresponding explosion slot. A bomb can only be placed on the rising edge of the bomb button, so holding the button does not repeatedly place bombs. A player may place a bomb only when alive, stationary, and when both the player's bomb and explosion slots are inactive.

After placement, the bomb timer is set to 53 ticks, which is about 1.66 seconds at 32 Hz. When the timer reaches zero, the bomb becomes a cross-shaped explosion covering five tiles: the center tile and the four adjacent tiles in the up, down, left, and right directions. The explosion lasts 32 ticks, or about one second.

Explosions can kill either player, including the player who placed the bomb. They do not trigger chain reactions and do not directly recolor territory tiles. If an explosion hits a soft wall, the wall is marked for destruction and is converted to a blue tile only after the explosion finishes. This delayed cleanup also handles the case where multiple explosions hit the same soft wall.

- **Death, Respawn, and End Game**

Death detection uses a 32 × 32-pixel rectangular overlap. If a player's rectangle overlaps any active explosion tile, the player is killed. When a player dies, the player becomes inactive, stops

moving, and is no longer rendered. All tiles owned by that player are reset to neutral blue tiles, and the player's score becomes zero.

A dead player respawns only after the explosion that caused the death has disappeared, and the spawn tile is safe. The spawn tile must be enterable, free of bombs, not covered by an active explosion, and not occupied or targeted by the other player. After respawn, the player returns to the original spawn position and immediately reclaims the spawn tile.

The game ends when at least one player reaches 50 points. If both players reach 50 in the same frame, the player with the higher score wins; if the scores are equal, the result is a tie. Restart has the highest priority and can be triggered by either controller at any time. When the restart is pressed, the map, players, bombs, explosions, scores, and sound states are reset for a new game.

3.2 HPS Interface

- **USB Controller**

The player interacts with GridBrawl through USB gamepad controllers connected to the HPS USB port. The FPGA does not directly read controller inputs or make gameplay decisions. Instead, the HPS runs a user-space USB controller module that reads controller packets, interprets button states, and converts them into high-level game commands.

In our implementation, the controller interface is handled by usbcontroller.c and usbcontroller.h. The software uses the libusb library to initialize the USB context, scan connected USB devices, and open gamepads with vendor ID 0x0079 and product ID 0x0011. After a matching controller is found, the program detaches the default kernel driver if necessary, claims interface 0, and reads input data from endpoint 0x81.

- **Communication Protocol**

The controller input is read as an 8-byte interrupt packet. The HPS polls the controller using libusb_interrupt_transfer() with a short timeout, allowing the main game loop to continue running even if no new packet is received during one polling attempt. When a valid 8-byte packet is received, the packet is parsed and stored as the cached input state for that controller.

Instead of passing raw USB bytes directly into the game logic, the controller module converts the packet into a controller_state_t structure. This structure contains six high-level control signals:

up

down

left

right

bomb

restart

The packet mapping used in our software is:

packet[3] == 0x00: move left

packet[3] == 0xff: move right

packet[4] == 0x00: move up

packet[4] == 0xff: move down

packet[5] bit 5: place bomb

packet[6] bit 5: restart game

If a read timeout occurs, the previous cached input state is kept. If the controller is disconnected or another USB error occurs, the corresponding device handle is closed and that controller state is cleared.

- **Integration with Game Logic**

The USB controller module does not directly update the game map, move players, place bombs, or determine collisions. Its role is only to translate low-level USB input packets into controller_state_t values.

During each frame, demo.c calls usb_read() to update the input states for the two players. If either player presses restart, demo.c clears the software audio queue before calling game_step(). Then game_step() uses the controller_state_t inputs to update movement, bomb placement, restart behavior, and the rest of the game state.

This separation keeps the controller layer independent from the game rules. The USB module answers what buttons are currently pressed, while game_logic.c decides what those button presses mean in the current game state.

4. Software/Hardware Interface

The HPS communicates with the FPGA hardware through the Avalon memory-mapped (Avalon-MM) bus. The HPS acts as an Avalon-MM master, while the FPGA exposes memory-mapped slave regions for the Tile Map RAM, Sprite Registers, and Audio Controller.

4.1 Address Mapping

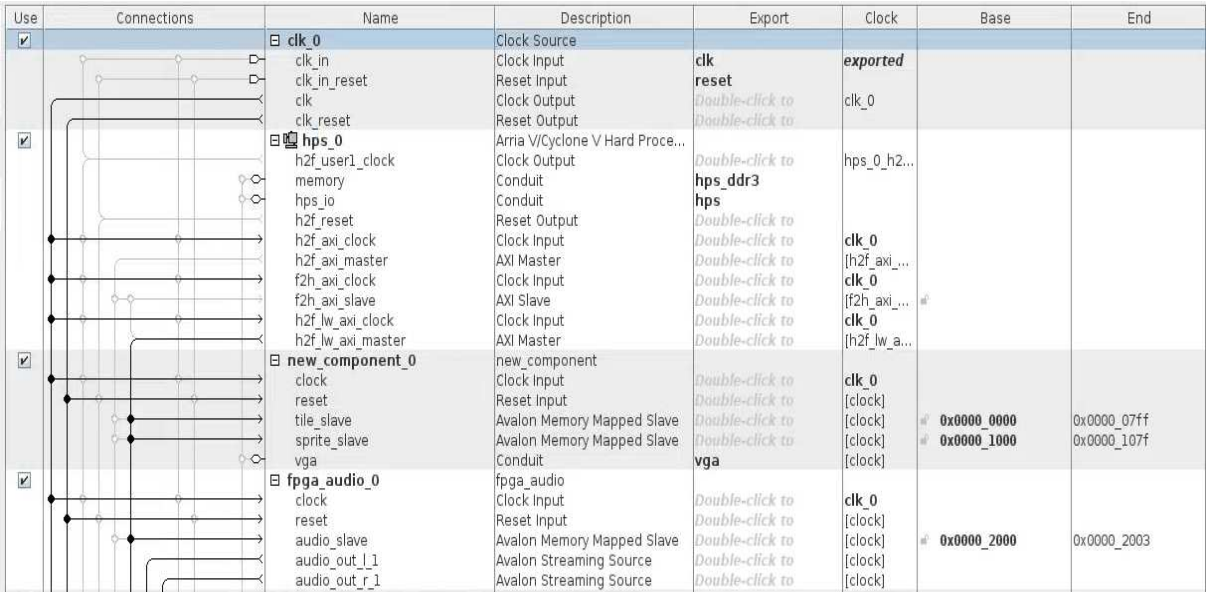

Figure 7. Platform Designer: HPS-to-FPGA Avalon-MM peripheral mapping.

The HPS communicates with the FPGA hardware modules through memory-mapped Avalon-MM interfaces. Each FPGA peripheral is assigned a dedicated physical address range within the lightweight HPS-to-FPGA bridge. The software accesses these hardware modules by writing to their corresponding memory-mapped registers.

The Tile Map RAM, Sprite Registers, and Audio Controller are mapped into separate address regions to simplify software control and avoid interface conflicts.

Base Address	Offset	Meaning
0xFF200000	0x000 - 0x4AC	Tile Map RAM, 300 entries, 4-byte-spaced write offsets
0xFF201000	0x000 - 0x074	Sprite 0 – Sprite 29 (4 bytes each)
0xFF202000	0x000 - 0x003	Audio Controller

This organization allows the HPS software to update graphics and audio state using lightweight register writes without directly generating VGA pixels or audio samples in software.

4.2 Detailed Register Definitions

All display-related states are stored in FPGA memory (Tile Map RAM and sprite registers) and updated by the HPS through the Avalon bus. The full game logic state is maintained in the HPS.

(1) Tile Map Memory

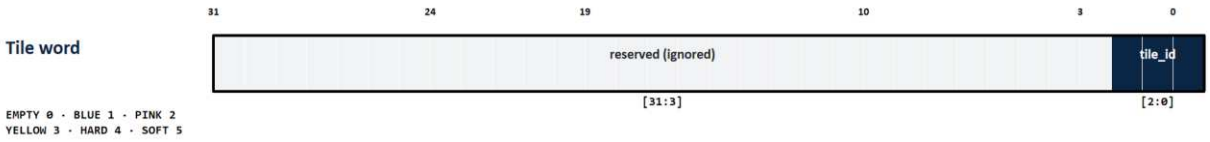

Figure 8. Tile word

Total tiles: 20 × 15 = 300. Each tile is represented by a 3-bit Tile ID in the FPGA Tile Map RAM.

Total theoretical storage: 300 × 3 = 900 bits ≈ 113 bytes

The HPS updates the tile map through the Avalon-MM interface. Although the software driver uses 32-bit MMIO writes for convenience and alignment, the hardware tile RAM only captures writedata[2:0]. The upper bits are ignored.

Each Tile Map entry corresponds to one 32 × 32-pixel grid cell on the 20 × 15 game map.

Bits [2:0]: Tile ID

- 0: Empty
- 1: Blue Tile
- 2: Pink Tile
- 3: Yellow Tile
- 4: Hard Wall
- 5: Soft Wall

Upper bits: Reserved and ignored by the FPGA tile RAM write path.

(2) Sprite Registers

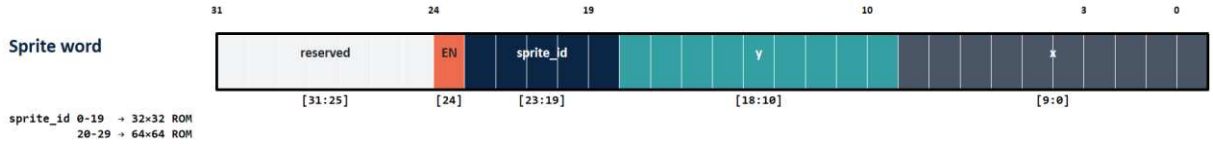

Figure 9. Sprite word

- Number of sprites: 30
- Each sprite: 32 bits

Total:

$$30 \times 32 = 960 \text{ bits} = 120 \text{ bytes}$$

Sprite Register Interface (0x000 - 0x074)

Dynamic display objects are controlled through memory-mapped sprite registers. Each sprite occupies one 32-bit register in the FPGA peripheral address space. The HPS updates these registers to control which sprites are visible and where they appear on the screen.

Each sprite register contains:

Field	Bits	Description
X position	[9:0]	0-639
Y position	[18:10]	0-479
Sprite ID	[23:19]	5-bit sprite image ID
Enable	[24]	visible or not
Reserved	[31:25]	Unused

The game uses 30 sprite registers for active on-screen objects. The 5-bit Sprite ID field selects the corresponding image stored in the sprite ROM.

The total size is 25 bits, and each sprite entry is aligned to 32 bits.

The X and Y fields specify the upper-left pixel position of the sprite on the VGA display. The Sprite ID selects the sprite image stored in the sprite image ROM. The Enable bit determines whether the sprite is rendered. When Enable is 0, the sprite is ignored by the rendering hardware.

(3) Audio Controller (0x000 – 0x003)

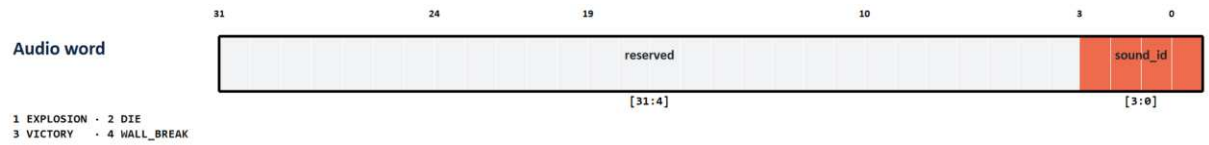

Figure 10. Audio word

The Audio Controller exposes one 32-bit write-only control register to the HPS. This register is used to trigger sound effects during gameplay. The HPS writes a sound event ID to this register whenever events such as explosions, player deaths, wall breaks, or victory occur.

- Bits [3:0]: Sound_ID

  0: None

  1: Explosion

  2: Player Dies

  3: Victory

  4: Wall Breaks

- Bits [31:4]: Reserved

The audio system continuously plays background music by default. When the HPS writes a valid, nonzero Sound_ID, the FPGA audio controller decodes the value and switches from background music playback to the selected sound effect. The sound effect is played once. After the sound effect reaches its end address in Audio ROM, the controller automatically resumes background music from its previous position.

The upper bits [31:4] are reserved and ignored by the hardware. For clean software behavior, these bits should be written as 0.

Write behavior:

Writing 0 has no effect and does not interrupt the current playback. Writing a valid value from 1 to 4 triggers the corresponding sound effect. If another sound effect is written while one is already playing, the new write updates the playback request and starts the newly selected sound effect.

This register is write-only from the software perspective. The HPS does not need to read the audio status because playback timing and return-to-background behavior are handled inside the FPGA audio controller.

5. Resource Budget

The following resource budget summarizes the estimated on-chip memory usage for the major graphics and audio assets stored in the FPGA design. The estimate focuses primarily on Tile ROMs, Sprite ROMs, and Audio ROM storage, since these structures dominate the custom hardware memory usage.

5.1 Video Resource Budget

The video subsystem uses palette-indexed graphics instead of full RGB pixel storage. Each pixel is stored as a compact palette index, which significantly reduces memory consumption while maintaining the required visual quality for the game.

Category	Graphics	Image Size	# of Images	Total Size (bits)
Player 1		32×32	4	32×32×4×6=24576
Player 2		32×32	4	32×32×4×6=24576
Bomb / Explosion		32×32	12	32×32×12×6=73728
Tiles		32×32	4	32×32×4×6=24576
Walls		32×32	2	32×32×2×6=12288

UI Assets		64×64	10	64×64×10×6=245760
Total	405504 bits			

5.2 Audio Resource Budget

The audio subsystem stores background music and sound effects as 8-bit mono PCM samples at an 8 kHz sample rate. This format provides acceptable audio quality while keeping the audio ROM size practical for FPGA implementation.

Category	Time (s)	Frequency (kHz)	# of Bits
Background Music	16	8	16×8000×8=1,024,000
Explosion	2	8	2×8000×8=128,000
Player Dies	1	8	1×8000×8=64,000
Victory	2	8	2×8000×8=128,000
Wall Breaks	1	8	1×8000×8=64,000

Total	1,408,000 bits

Overall, the estimated graphics and audio storage requirements fit within the available FPGA on-chip memory resources. The compact graphics and audio formats allow the system to support real-time rendering and audio playback without requiring external memory for asset storage.

6. Testing and Validating

The complete game system was validated on the DE1-SoC platform using the final FPGA hardware design and HPS software runtime. The VGA output was verified on a 640 × 480 display, including tile rendering, sprite overlay, gameplay UI display, and real-time map updates from the HPS. The scanline-based renderer and double-buffered line buffer maintained stable VGA output without visible tearing artifacts during gameplay.

Hardware/software communication was validated through the Avalon-MM interface. The HPS software successfully updated Tile Map RAM entries, Sprite Registers, and Audio Controller registers through memory-mapped writes. These updates were correctly reflected by the FPGA rendering and audio pipelines in real time.

The audio subsystem was validated using both background music playback and event-triggered sound effects. Sound effects such as explosions and victory playback were triggered through Sound_ID writes from the HPS and played through the Intel Audio IP and WM8731 audio CODEC. The FPGA audio controller correctly handled playback switching and background music recovery after sound-effect completion.

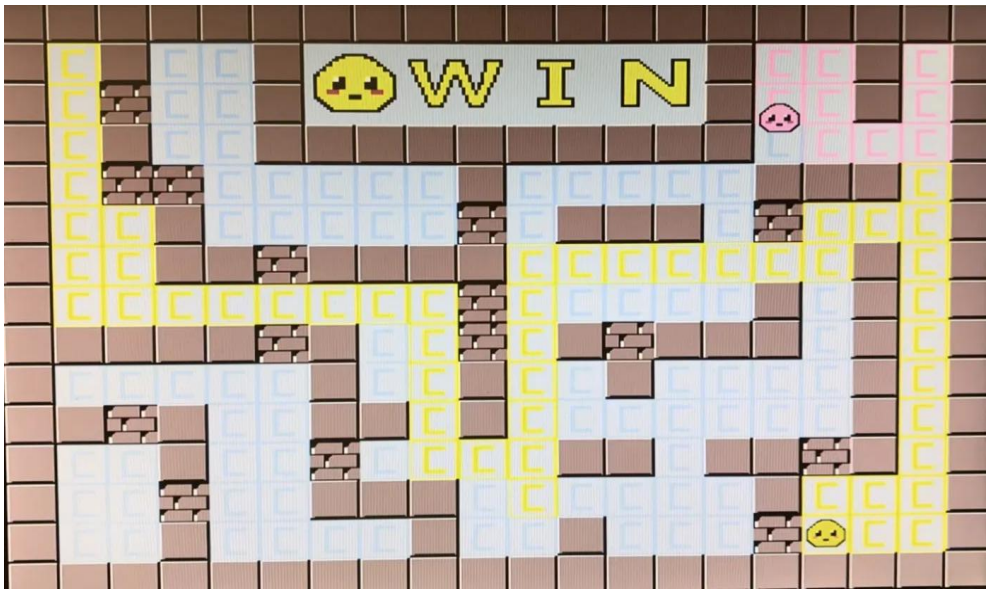

Figure11. FPGA-generated VGA output

7. Challenges and Lessons Learned

• Real-time VGA rendering synchronization:

The FPGA video pipeline must continuously generate pixels at the VGA timing rate, while the HPS updates game state at a much lower game-loop rate. This created a synchronization boundary between persistent game state stored in Tile Map RAM and Sprite Registers, and transient pixel data generated dynamically through the scanline rendering pipeline.

• Line buffer timing constraints:

The line buffer architecture required strict rendering order and timing control. Background tile pixels must be generated first, followed by sprite pixels with transparency handling, before the final scanline is sent to the VGA DAC. Since sprite pixels can overwrite background pixels while transparent pixels must be ignored, the rendering pipeline depends on deterministic pipeline ordering and fixed ROM access latency.

• FPGA memory optimization:

Graphics assets were stored as palette-indexed ROM data instead of full RGB pixels, with palette index 0 reserved for transparent sprite pixels. This significantly reduced the amount of on-chip memory required for tiles, sprites, and UI assets. Similarly, audio was stored as 8-bit mono PCM data at an 8 kHz sample rate to keep audio ROM usage practical for FPGA implementation.

• Real-time audio timing:

The audio subsystem demonstrated the importance of matching stored sample timing with codec timing requirements. Since the WM8731 interface generates sample-ready events at a higher rate than the stored PCM sample rate, the FPGA audio controller uses a sample-hold mechanism to preserve the intended playback speed.

• Hardware/software co-design:

The Avalon-MM interface was a key part of the system architecture. The HPS handles USB controller input, game rules, collision logic, scoring, and high-level state updates, while the FPGA handles deterministic VGA rendering and audio playback. This partitioning kept software flexible while keeping time-critical video and audio output in hardware.

• Modular hardware architecture:

The project demonstrated that FPGA game design is fundamentally a synchronization and resource-management problem. Modular hardware blocks for tile loading, sprite loading, palette conversion, line buffering, and audio playback made system integration and timing verification significantly more manageable.

8. Code

8.1 Hardware

- tile-loader.sv

```
module tile_loader (
    input  logic        clk,
    input  logic        reset,

    input  logic        start,       // pulse: start generating one scanline
    input  logic [9:0]  vcount,      // current scanline, 0~479

    // Tile Map RAM read side
    output logic [8:0]  tilemap_rdaddr,
    input  logic [2:0]  tile_id,
```

```
    // Tile Image ROM
    output logic [12:0] tile_rom_addr,
    input  logic [5:0]  tile_pixel_index,

    // Background pixel stream to sprite_loader
    output logic        bg_valid,
    output logic [9:0]  bg_x,
    output logic [5:0]  bg_pixel_index,

    output logic        busy,
    output logic        done
);

    typedef enum logic [2:0] {
        S_IDLE,
        S_TILE_ADDR,
        S_TILE_WAIT,
        S_TILE_CAPTURE,
        S_PIXELS,
        S_DONE
    } state_t;

    state_t state;

    logic [3:0] tile_y;        // 0~14
    logic [4:0] row_in_tile;   // 0~31

    logic [4:0] tile_x;        // 0~19
    logic [5:0] pix_col;       // 0~32, 32 means drain last ROM output
```

```
logic [2:0] cur_tile_id;

logic       rom_valid_d;
logic [9:0]  rom_x_d;

function automatic [8:0] tilemap_addr_calc (
    input logic [3:0] ty,
    input logic [4:0] tx
);
    // ty * 20 + tx = ty * 16 + ty * 4 + tx
    tilemap_addr_calc = ({5'b0, ty} << 4) + ({5'b0, ty} << 2) + {4'b0, tx};
endfunction

assign busy = (state != S_IDLE);

always_ff @(posedge clk) begin
    if (reset) begin
        state          <= S_IDLE;
        tilemap_rdaddr <= 9'd0;
        tile_rom_addr  <= 13'd0;

        tile_y         <= 4'd0;
        row_in_tile    <= 5'd0;
        tile_x         <= 5'd0;
        pix_col        <= 6'd0;
        cur_tile_id    <= 3'd0;

        rom_valid_d    <= 1'b0;
        rom_x_d        <= 10'd0;

        bg_valid       <= 1'b0;
```

```
    bg_x          <= 10'd0;
    bg_pixel_index <= 6'd0;
    done          <= 1'b0;
end else begin
    bg_valid <= 1'b0;
    done     <= 1'b0;

    case (state)
        S_IDLE: begin
            rom_valid_d <= 1'b0;

            if (start) begin
                tile_y      <= vcount[8:5];   // vcount / 32
                row_in_tile <= vcount[4:0];   // vcount % 32
                tile_x      <= 5'd0;
                pix_col     <= 6'd0;
                state       <= S_TILE_ADDR;
            end
        end

        S_TILE_ADDR: begin
            tilemap_rdaddr <= tilemap_addr_calc(tile_y, tile_x);
            state          <= S_TILE_WAIT;
        end

        // Tile Map RAM has registered output, so wait one cycle
        S_TILE_WAIT: begin
            state <= S_TILE_CAPTURE;
        end

        S_TILE_CAPTURE: begin
```

```
    cur_tile_id <= tile_id;
    pix_col     <= 6'd0;

    // Issue first ROM address on next S_PIXELS cycle
    rom_valid_d <= 1'b0;
    state       <= S_PIXELS;
end

S_PIXELS: begin
    // Output previous ROM result
    bg_valid       <= rom_valid_d;
    bg_x           <= rom_x_d;
    bg_pixel_index <= tile_pixel_index;

    if (pix_col < 6'd32) begin
        // Issue current pixel address to Tile Image ROM
        tile_rom_addr <= {
            cur_tile_id,        // 3 bits
            row_in_tile,        // 5 bits
            pix_col[4:0]        // 5 bits
        };

        rom_x_d     <= (tile_x * 10'd32) + pix_col[4:0];
        rom_valid_d <= 1'b1;
        pix_col     <= pix_col + 6'd1;
    end else begin
        // pix_col == 32: drain the last ROM output this cycle
        rom_valid_d <= 1'b0;

        if (tile_x == 5'd19) begin
            state <= S_DONE;
```

```
                end else begin
                    tile_x  <= tile_x + 5'd1;
                    pix_col <= 6'd0;
                    state   <= S_TILE_ADDR;
                end
            end
        end

        S_DONE: begin
            done  <= 1'b1;
            state <= S_IDLE;
        end

        default: begin
            state <= S_IDLE;
        end
      endcase
    end
  end

endmodule
```

- Sprite_loader.sv

```
module sprite_loader (
  input  logic        clk,
  input  logic        reset,

  input  logic [9:0]  vcount,

  // Sprite registers from sprite_registers module
  input  logic [31:0] sprite_regs [0:29],
```

```
    // Background pixel stream from tile_loader
    input  logic        bg_valid,
    input  logic [9:0]  bg_x,
    input  logic [5:0]  bg_pixel_index,
    input  logic        bg_done,

    // 32x32 Sprite Image ROM
    output logic [14:0] sprite32_rom_addr,
    input  logic [5:0]  sprite32_pixel_index,

    // 64x64 Sprite Image ROM
    output logic [15:0] sprite64_rom_addr,
    input  logic [5:0]  sprite64_pixel_index,

    // Final pixel stream to linebuffer_controller
    output logic        out_valid,
    output logic [9:0]  out_x,
    output logic [5:0]  out_pixel_index,

    output logic        busy,
    output logic        done
);

    localparam logic [5:0] TRANSPARENT_INDEX = 6'd0;

    typedef enum logic [2:0] {
        S_IDLE,
        S_BACKGROUND,
        S_SPRITE_SETUP,
        S_SPRITE_PIXELS,
```

```
    S_NEXT_SPRITE,
    S_DONE
} state_t;

state_t state;

logic [4:0] sprite_idx;
logic [4:0] sprite_reg_idx;

logic       cur_enable;
logic [4:0]  cur_id;
logic [9:0]  cur_x;
logic [9:0]  cur_y;
logic [6:0]  cur_w;
logic [6:0]  cur_h;
logic [6:0]  cur_col;
logic [6:0]  cur_row;
logic       cur_use32;

logic       rom_valid_d;
logic       rom_write_d;
logic       rom_use32_d;
logic [9:0]  rom_x_d;
logic       rom_valid_q;
logic       rom_write_q;
logic       rom_use32_q;
logic [9:0]  rom_x_q;
logic       rom_valid_r;
logic       rom_write_r;
logic       rom_use32_r;
logic [9:0]  rom_x_r;
```

```
logic [9:0]  cur_col_x;
logic [9:0]  cur_y_end;
logic [4:0]  local64_id;

assign cur_col_x = {3'b0, cur_col};
assign cur_y_end = cur_y + {3'b0, cur_h};
assign local64_id = cur_id - 5'd20;

assign busy = (state != S_IDLE);

function automatic logic [6:0] sprite_size(input logic [4:0] sid);
   begin
      if (sid < 5'd20)
         sprite_size = 7'd32;
      else
         sprite_size = 7'd64;
   end
endfunction

function automatic logic [4:0] render_sprite_index(input logic [4:0] draw_idx);
   begin
      case (draw_idx)
         5'd0: render_sprite_index = 5'd0;
         5'd1: render_sprite_index = 5'd1;
         5'd2: render_sprite_index = 5'd2;
         5'd3: render_sprite_index = 5'd3;
         5'd4: render_sprite_index = 5'd18;
         5'd5: render_sprite_index = 5'd19;
         default: begin
            if (draw_idx < 5'd20)
```

```
                render_sprite_index = draw_idx - 5'd2;
            else
                render_sprite_index = draw_idx;
        end
    endcase
  end
endfunction

assign sprite_reg_idx = render_sprite_index(sprite_idx);

always_ff @(posedge clk) begin
  if (reset) begin
    state             <= S_IDLE;

    sprite_idx        <= 5'd0;

    cur_enable        <= 1'b0;
    cur_id            <= 5'd0;
    cur_x             <= 10'd0;
    cur_y             <= 10'd0;
    cur_w             <= 7'd0;
    cur_h             <= 7'd0;
    cur_col           <= 7'd0;
    cur_row           <= 7'd0;
    cur_use32         <= 1'b1;

    rom_valid_d       <= 1'b0;
    rom_write_d       <= 1'b0;
    rom_use32_d       <= 1'b1;
    rom_x_d           <= 10'd0;
    rom_valid_q       <= 1'b0;
```

```
    rom_write_q       <= 1'b0;
    rom_use32_q       <= 1'b1;
    rom_x_q           <= 10'd0;
    rom_valid_r       <= 1'b0;
    rom_write_r       <= 1'b0;
    rom_use32_r       <= 1'b1;
    rom_x_r           <= 10'd0;

    sprite32_rom_addr  <= 15'd0;
    sprite64_rom_addr  <= 16'd0;

    out_valid         <= 1'b0;
    out_x             <= 10'd0;
    out_pixel_index   <= 6'd0;
    done              <= 1'b0;
end else begin
    out_valid <= 1'b0;
    done      <= 1'b0;

    case (state)

        S_IDLE: begin
            rom_valid_d <= 1'b0;
            rom_write_d <= 1'b0;
            rom_valid_q <= 1'b0;
            rom_write_q <= 1'b0;
            rom_valid_r <= 1'b0;
            rom_write_r <= 1'b0;

            if (bg_valid) begin
                out_valid      <= 1'b1;
```

```
        out_x         <= bg_x;
        out_pixel_index <= bg_pixel_index;
        state         <= S_BACKGROUND;
    end
end

S_BACKGROUND: begin
    if (bg_valid) begin
        out_valid     <= 1'b1;
        out_x         <= bg_x;
        out_pixel_index <= bg_pixel_index;
    end

    if (bg_done) begin
        sprite_idx <= 5'd0;
        state      <= S_SPRITE_SETUP;
    end
end

S_SPRITE_SETUP: begin
    cur_enable <= sprite_regs[sprite_reg_idx][24];
    cur_id     <= sprite_regs[sprite_reg_idx][23:19];
    cur_y      <= {1'b0, sprite_regs[sprite_reg_idx][18:10]};
    cur_x      <= sprite_regs[sprite_reg_idx][9:0];
    cur_w      <= sprite_size(sprite_regs[sprite_reg_idx][23:19]);
    cur_h      <= sprite_size(sprite_regs[sprite_reg_idx][23:19]);
    cur_use32  <= (sprite_regs[sprite_reg_idx][23:19] < 5'd20);

    if (vcount >= {1'b0, sprite_regs[sprite_reg_idx][18:10]})
        cur_row <= vcount[6:0] - sprite_regs[sprite_reg_idx][16:10];
    else
```

```
            cur_row <= 7'd0;

        cur_col     <= 7'd0;
        rom_valid_d <= 1'b0;
        rom_write_d <= 1'b0;
        rom_valid_q <= 1'b0;
        rom_write_q <= 1'b0;
        rom_valid_r <= 1'b0;
        rom_write_r <= 1'b0;

        if (sprite_regs[sprite_reg_idx][24] &&
            (vcount >= {1'b0, sprite_regs[sprite_reg_idx][18:10]}) &&
            (vcount < ({1'b0, sprite_regs[sprite_reg_idx][18:10]} +
                    {3'b0, sprite_size(sprite_regs[sprite_reg_idx][23:19])})) &&
            (sprite_regs[sprite_reg_idx][9:0] < 10'd640)) begin
            state <= S_SPRITE_PIXELS;
        end else begin
            if (sprite_idx == 5'd29) begin
                sprite_idx <= 5'd0;
                state      <= S_DONE;
            end else begin
                sprite_idx <= sprite_idx + 5'd1;
                state      <= S_SPRITE_SETUP;
            end
        end
    end

    S_SPRITE_PIXELS: begin
        // Output previous ROM result
        if (rom_valid_r && rom_write_r) begin
            if (rom_use32_r) begin
```

```
        if (sprite32_pixel_index != TRANSPARENT_INDEX) begin
            out_valid       <= 1'b1;
            out_x           <= rom_x_r;
            out_pixel_index <= sprite32_pixel_index;
        end
    end else begin
        if (sprite64_pixel_index != TRANSPARENT_INDEX) begin
            out_valid       <= 1'b1;
            out_x           <= rom_x_r;
            out_pixel_index <= sprite64_pixel_index;
        end
    end
end

rom_valid_r <= rom_valid_q;
rom_write_r <= rom_write_q;
rom_use32_r <= rom_use32_q;
rom_x_r     <= rom_x_q;
rom_valid_q <= rom_valid_d;
rom_write_q <= rom_write_d;
rom_use32_q <= rom_use32_d;
rom_x_q     <= rom_x_d;

if (cur_col < cur_w) begin
    rom_valid_d <= 1'b1;
    rom_use32_d <= cur_use32;
    rom_x_d     <= cur_x + cur_col_x;

    if ((cur_x + cur_col_x) < 10'd640) begin

        rom_write_d <= 1'b1;
```

```
            if (cur_use32) begin
               sprite32_rom_addr <= ({10'd0, cur_id} << 10) +
                                    ({10'd0, cur_row[4:0]} << 5) +
                                    {10'd0, cur_col[4:0]};
               sprite64_rom_addr <= 16'd0;
            end else begin
               sprite64_rom_addr <= ({12'd0, local64_id[3:0]} << 12) +
                                    ({10'd0, cur_row[5:0]} << 6) +
                                    {10'd0, cur_col[5:0]};
               sprite32_rom_addr <= 15'd0;
            end
         end else begin
            rom_write_d       <= 1'b0;
            sprite32_rom_addr <= 15'd0;
            sprite64_rom_addr <= 16'd0;
         end

         cur_col <= cur_col + 7'd1;
      end else begin
         rom_valid_d <= 1'b0;
         rom_write_d <= 1'b0;

         if (!rom_valid_d && !rom_valid_q && !rom_valid_r) begin
            state <= S_NEXT_SPRITE;
         end
      end
   end

   S_NEXT_SPRITE: begin
      if (sprite_idx == 5'd29) begin
```

```
                sprite_idx <= 5'd0;
                state      <= S_DONE;
            end else begin
                sprite_idx <= sprite_idx + 5'd1;
                state      <= S_SPRITE_SETUP;
            end
        end

        S_DONE: begin
            done  <= 1'b1;
            state <= S_IDLE;
        end

        default: begin
            state <= S_IDLE;
        end

    endcase
  end
end

endmodule
```

- Sprite_register

```
module sprite_registers (
  input  logic        clk,
  input  logic        reset,

  input  logic        chipselect,
  input  logic        write,
  input  logic [4:0]   address,      // sprite index: 0~29
```

```
    input  logic [31:0]  writedata,

    output logic [31:0]  sprite_regs [0:29]
);

    integer i;

    always_ff @(posedge clk) begin
        if (reset) begin
            for (i = 0; i < 30; i = i + 1)
                sprite_regs[i] <= 32'b0;
        end else begin
            if (chipselect && write && (address < 5'd30)) begin
                sprite_regs[address] <= {
                    7'b0,             // [31:25] reserved, forced to 0
                    writedata[24],    // [24] enable
                    writedata[23:19],  // [23:19] sprite_id
                    writedata[18:10],  // [18:10] y position
                    writedata[9:0]    // [9:0] x position
                };
            end
        end
    end

endmodule
```

- Linebuffer_controller.sv

```
module linebuffer_controller (
    input  logic        clk,
    input  logic        reset,
```

```
    // swap front/back buffer at a safe line boundary
    input  logic        swap_buffers,

    // write side: from sprite_loader
    input  logic        wr_valid,
    input  logic [9:0]  wr_x,            // 0~639
    input  logic [5:0]  wr_pixel_index,   // palette index

    // read side: from VGA counter
    input  logic [9:0]  hcount,          // real pixel x, 0~639
    input  logic        visible_area,

    // output to palette
    output logic [5:0]  pixel_index_to_palette,

    output logic        front_buffer_select
);

    // ============================================================
    // One physical RAM is split into two 640-pixel line buffers:
    //   address 0   ~ 639   : buffer 0
    //   address 640 ~ 1279  : buffer 1
    // front_buffer_select = 0:
    //   VGA reads buffer 0, renderer writes buffer 1
    // front_buffer_select = 1:
    //   VGA reads buffer 1, renderer writes buffer 0
    // ============================================================

    localparam logic [10:0] LINE_OFFSET = 11'd640;

    logic [10:0] rd_addr;
```

```
logic [10:0] wr_addr;
logic [5:0]  ram_q;

logic [1:0] visible_area_pipe;

// ------------------------------------------------------------
// Front/back buffer select
// ------------------------------------------------------------

always_ff @(posedge clk) begin
   if (reset) begin
      front_buffer_select <= 1'b0;
      visible_area_pipe   <= 2'b00;
   end else begin
      visible_area_pipe <= {visible_area_pipe[0], visible_area};

      if (swap_buffers) begin
         front_buffer_select <= ~front_buffer_select;
      end
   end
end

// ------------------------------------------------------------
// Address generation
// ------------------------------------------------------------
//
// If front_buffer_select = 0:
//   read  buffer 0: address hcount
//   write buffer 1: address 640 + wr_x
//
// If front_buffer_select = 1:
```

```
//   read  buffer 1: address 640 + hcount
//   write buffer 0: address wr_x
// ------------------------------------------------------------

always_comb begin
    if (front_buffer_select == 1'b0) begin
        rd_addr = {1'b0, hcount};
        wr_addr = LINE_OFFSET + {1'b0, wr_x};
    end else begin
        rd_addr = LINE_OFFSET + {1'b0, hcount};
        wr_addr = {1'b0, wr_x};
    end
end

// ------------------------------------------------------------
// 1280 x 6 dual-port line buffer RAM
// ------------------------------------------------------------

linebuffer_ram linebuffer_ram_inst (
    .clock     (clk),
    .data      (wr_pixel_index),
    .rdaddress (rd_addr),
    .wraddress (wr_addr),
    .wren      (wr_valid),
    .q         (ram_q)
);

// ------------------------------------------------------------
// Output to palette
// ------------------------------------------------------------
```

```
    always_comb begin
        if (visible_area_pipe[1]) begin
            pixel_index_to_palette = ram_q;
        end else begin
            pixel_index_to_palette = 6'd0;
        end
    end

endmodule
```

- Video_generator_top.sv

```
module video_generator_top (
    input  logic        clk,
    input  logic        reset,

    // Tile Map RAM write interface
    input  logic        tile_chipselect,
    input  logic        tile_write,
    input  logic [8:0]  tile_address,
    input  logic [31:0] tile_writedata,

    // Sprite Registers write interface
    input  logic        sprite_chipselect,
    input  logic        sprite_write,
    input  logic [4:0]  sprite_address,
    input  logic [31:0] sprite_writedata,

    // Full VGA output to VGA DAC
    output logic [7:0]  VGA_R,
    output logic [7:0]  VGA_G,
    output logic [7:0]  VGA_B,
```

```
    output logic        VGA_CLK,
    output logic        VGA_HS,
    output logic        VGA_VS,
    output logic        VGA_BLANK_n,
    output logic        VGA_SYNC_n,

    // Debug
    output logic        tile_loader_busy,
    output logic        tile_loader_done,
    output logic        front_buffer_select
);

    // ============================================================
    // VGA counters
    // ============================================================

    logic [10:0] hcount_raw;
    logic [9:0]  vcount;
    logic [9:0]  pixel_x;
    logic        visible_area;

    vga_counters counters (
        .clk50       (clk),
        .reset       (reset),

        .hcount      (hcount_raw),
        .vcount      (vcount),

        .VGA_CLK     (VGA_CLK),
        .VGA_HS      (VGA_HS),
        .VGA_VS      (VGA_VS),
```

```
    .VGA_BLANK_n (VGA_BLANK_n),
    .VGA_SYNC_n  (VGA_SYNC_n)
);

assign pixel_x      = hcount_raw[10:1];
assign visible_area = VGA_BLANK_n;

// ============================================================
// Render control
// ============================================================

localparam logic [10:0] H_TOTAL_50MHZ = 11'd1600;
localparam logic [9:0]  V_ACTIVE      = 10'd480;

logic       line_begin;
logic       line_end;
logic       render_start;
logic       swap_buffers;
logic [9:0]  render_vcount;
logic       render_done_pending;

logic       sprite_loader_busy;
logic       sprite_loader_done;

typedef enum logic [1:0] {
    R_IDLE,
    R_RENDER
} render_state_t;

render_state_t render_state;
```

```
assign line_begin = (hcount_raw == 11'd0);
assign line_end   = (hcount_raw == (H_TOTAL_50MHZ - 11'd1));

always_ff @(posedge clk) begin
   if (reset) begin
      render_start       <= 1'b0;
      swap_buffers       <= 1'b0;
      render_vcount      <= 10'd0;
      render_done_pending <= 1'b0;
      render_state       <= R_IDLE;
   end else begin
      render_start       <= 1'b0;
      swap_buffers       <= 1'b0;

      if (line_end && render_done_pending) begin
         swap_buffers       <= 1'b1;
         render_done_pending <= 1'b0;
      end

      case (render_state)
         R_IDLE: begin
            // During active video, prepare the next visible line.
            // On line 479, prepare line 0 for the next frame.
            if (line_begin && (vcount < V_ACTIVE)) begin
               if (vcount == (V_ACTIVE - 10'd1))
                  render_vcount <= 10'd0;
               else
                  render_vcount <= vcount + 10'd1;

               render_start <= 1'b1;
               render_state <= R_RENDER;
```

```
            end
        end

        R_RENDER: begin
            if (sprite_loader_done) begin
                render_done_pending <= 1'b1;
                render_state        <= R_IDLE;

                // If the sprite renderer finishes exactly on the safe
                // boundary, allow the swap immediately.
                if (line_end) begin
                    swap_buffers        <= 1'b1;
                    render_done_pending <= 1'b0;
                end
            end
        end

        default: begin
            render_state <= R_IDLE;
        end
      endcase
   end
end

// ============================================================
// Tile Map RAM
// ============================================================

logic [8:0] tilemap_rdaddr;
logic [2:0] tile_id;
```

```
tilemap_ram tilemap_ram_inst (
    .clock     (clk),
    .data      (tile_writedata[2:0]),
    .wraddress (tile_address),
    .wren      (tile_chipselect && tile_write && (tile_address < 9'd300)),
    .rdaddress (tilemap_rdaddr),
    .q         (tile_id)
);

// ============================================================
// Sprite Registers
// ============================================================

logic [31:0] sprite_regs [0:29];

sprite_registers sprite_registers_inst (
    .clk         (clk),
    .reset       (reset),

    .chipselect  (sprite_chipselect),
    .write       (sprite_write),
    .address     (sprite_address),
    .writedata   (sprite_writedata),

    .sprite_regs (sprite_regs)
);

// ============================================================
// Tile Image ROM
// ============================================================
```

```
logic [12:0] tile_rom_addr;
logic [5:0]  tile_pixel_index;

tile_image_rom tile_image_rom_inst (
    .address (tile_rom_addr),
    .clock   (clk),
    .q       (tile_pixel_index)
);

// ============================================================
// Tile Loader
// ============================================================

logic        bg_valid;
logic [9:0]  bg_x;
logic [5:0]  bg_pixel_index;

tile_loader tile_loader_inst (
    .clk             (clk),
    .reset           (reset),

    .start           (render_start),
    .vcount          (render_vcount),

    .tilemap_rdaddr   (tilemap_rdaddr),
    .tile_id          (tile_id),

    .tile_rom_addr    (tile_rom_addr),
    .tile_pixel_index (tile_pixel_index),

    .bg_valid         (bg_valid),
```

```
    .bg_x           (bg_x),
    .bg_pixel_index   (bg_pixel_index),

    .busy           (tile_loader_busy),
    .done           (tile_loader_done)
);

// ============================================================
// Sprite Image ROMs
// ============================================================

logic [14:0] sprite32_rom_addr;
logic [5:0]  sprite32_pixel_index;

logic [15:0] sprite64_rom_addr;
logic [5:0]  sprite64_pixel_index;

sprite32_image_rom sprite32_image_rom_inst (
    .address (sprite32_rom_addr),
    .clock   (clk),
    .q       (sprite32_pixel_index)
);

sprite64_image_rom sprite64_image_rom_inst (
    .address (sprite64_rom_addr),
    .clock   (clk),
    .q       (sprite64_pixel_index)
);

// ============================================================
// Sprite Loader / Overlay Renderer
```

```
// ============================================================
logic       sprite_wr_valid;
logic [9:0]  sprite_wr_x;
logic [5:0]  sprite_wr_pixel_index;

sprite_loader sprite_loader_inst (
    .clk                (clk),
    .reset              (reset),

    .vcount             (render_vcount),

    .sprite_regs        (sprite_regs),

    .bg_valid           (bg_valid),
    .bg_x               (bg_x),
    .bg_pixel_index     (bg_pixel_index),
    .bg_done            (tile_loader_done),

    .sprite32_rom_addr    (sprite32_rom_addr),
    .sprite32_pixel_index (sprite32_pixel_index),

    .sprite64_rom_addr    (sprite64_rom_addr),
    .sprite64_pixel_index (sprite64_pixel_index),

    .out_valid          (sprite_wr_valid),
    .out_x              (sprite_wr_x),
    .out_pixel_index    (sprite_wr_pixel_index),

    .busy               (sprite_loader_busy),
    .done               (sprite_loader_done)
);
```

```
// ============================================================
// Line Buffer Controller
// ============================================================

logic [5:0] color_index;

linebuffer_controller linebuffer_controller_inst (
    .clk                (clk),
    .reset              (reset),

    .swap_buffers       (swap_buffers),

    .wr_valid           (sprite_wr_valid),
    .wr_x               (sprite_wr_x),
    .wr_pixel_index     (sprite_wr_pixel_index),

    .hcount             (pixel_x),
    .visible_area       (visible_area),

    .pixel_index_to_palette (color_index),
    .front_buffer_select    (front_buffer_select)
);

// ============================================================
// Palette
// ============================================================

logic [23:0] rgb888;

palette palette_inst (
```

```
        .color_index (color_index),
        .rgb         (rgb888)
    );

    // ============================================================
    // VGA RGB output
    // ============================================================
    //
    // visible_area blanking is already handled inside linebuffer_controller:
    // outside visible area, it outputs color_index = 0.
    //

    always_comb begin
        VGA_R = rgb888[23:16];
        VGA_G = rgb888[15:8];
        VGA_B = rgb888[7:0];
    end

endmodule



// ============================================================
// VGA counters
// ============================================================

module vga_counters (
    input  logic        clk50,
    input  logic        reset,

    output logic [10:0] hcount,
    output logic [9:0]  vcount,
```

```
    output logic        VGA_CLK,
    output logic        VGA_HS,
    output logic        VGA_VS,
    output logic        VGA_BLANK_n,
    output logic        VGA_SYNC_n
);

    parameter HACTIVE      = 11'd1280,
              HFRONT_PORCH = 11'd32,
              HSYNC        = 11'd192,
              HBACK_PORCH  = 11'd96,
              HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH;

    parameter VACTIVE      = 10'd480,
              VFRONT_PORCH = 10'd10,
              VSYNC        = 10'd2,
              VBACK_PORCH  = 10'd33,
              VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC + VBACK_PORCH;

    logic endOfLine;
    logic endOfField;

    always_ff @(posedge clk50 or posedge reset) begin
        if (reset)
            hcount <= 11'd0;
        else if (endOfLine)
            hcount <= 11'd0;
        else
            hcount <= hcount + 11'd1;
    end
```

```
    assign endOfLine = (hcount == HTOTAL - 11'd1);

    always_ff @(posedge clk50 or posedge reset) begin
        if (reset)
            vcount <= 10'd0;
        else if (endOfLine) begin
            if (endOfField)
                vcount <= 10'd0;
            else
                vcount <= vcount + 10'd1;
        end
    end

    assign endOfField = (vcount == VTOTAL - 10'd1);

    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;

    assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
                         !( vcount[9] | (vcount[8:5] == 4'b1111) );

    assign VGA_CLK = hcount[0];

endmodule
```

- fpga_audio.sv

```
`define BGM_BEGIN 18'h00000
`define BGM_END   18'h1F3FF

module fpga_audio (
    input  logic        clk,
    input  logic        reset,

    input  logic        left_chan_ready,
    input  logic        right_chan_ready,

    input  logic [31:0] writedata,
    input  logic        write,
    input  logic        chipselect,

    output logic [15:0] sample_data_l,
    output logic        sample_valid_l,
    output logic [15:0] sample_data_r,
    output logic        sample_valid_r
);

    localparam logic [3:0] SOUND_NONE       = 4'd0;
    localparam logic [3:0] SOUND_EXPLOSION  = 4'd1;
    localparam logic [3:0] SOUND_PLAYER_DIE = 4'd2;
    localparam logic [3:0] SOUND_VICTORY    = 4'd3;
    localparam logic [3:0] SOUND_WALL_BREAK = 4'd4;

    localparam logic [17:0] EXP_BEGIN     = 18'h1F400;
    localparam logic [17:0] EXP_END       = 18'h2327F;

    localparam logic [17:0] DIE_BEGIN     = 18'h23280;
    localparam logic [17:0] DIE_END       = 18'h251BF;
```

```
localparam logic [17:0] VICTORY_BEGIN = 18'h251C0;
localparam logic [17:0] VICTORY_END   = 18'h2903F;

localparam logic [17:0] WALL_BEGIN    = 18'h29040;
localparam logic [17:0] WALL_END      = 18'h2AF7F;

typedef enum logic [0:0] {
    PLAY_BGM,
    PLAY_SFX
} play_state_t;

play_state_t state;

logic start_sfx_write;
logic audio_ready;
logic audio_ready_d;
logic audio_ready_pulse;

logic [17:0] sound_address;
logic [17:0] sound_end_address;
logic [17:0] bgm_address;

logic [2:0] sample_hold_counter;

logic [7:0] sound_data;

audio_rom audio_rom_inst (
    .address (sound_address),
    .clock   (clk),
    .q       (sound_data)
```

```
);

assign sample_data_l = {sound_data ^ 8'h80, 8'b0};
assign sample_data_r = {sound_data ^ 8'h80, 8'b0};

assign audio_ready = left_chan_ready && right_chan_ready;
assign audio_ready_pulse = audio_ready && !audio_ready_d;

assign start_sfx_write = chipselect && write &&
                         (writedata[3:0] >= SOUND_EXPLOSION) &&
                         (writedata[3:0] <= SOUND_WALL_BREAK);

always_ff @(posedge clk) begin
   if (reset) begin
      state            <= PLAY_BGM;

      sound_address     <= `BGM_BEGIN;
      sound_end_address <= `BGM_END;
      bgm_address       <= `BGM_BEGIN;
      sample_hold_counter <= 3'd0;
      audio_ready_d     <= 1'b0;

      sample_valid_l    <= 1'b0;
      sample_valid_r    <= 1'b0;
   end else begin
      audio_ready_d <= audio_ready;

      sample_valid_l <= 1'b0;
      sample_valid_r <= 1'b0;

      if (chipselect && write) begin
```

```
case (writedata[3:0])
    SOUND_EXPLOSION: begin
        state           <= PLAY_SFX;
        sound_address     <= EXP_BEGIN;
        sound_end_address <= EXP_END;
        sample_hold_counter <= 3'd0;
    end

    SOUND_PLAYER_DIE: begin
        state           <= PLAY_SFX;
        sound_address     <= DIE_BEGIN;
        sound_end_address <= DIE_END;
        sample_hold_counter <= 3'd0;
    end

    SOUND_VICTORY: begin
        state           <= PLAY_SFX;
        sound_address     <= VICTORY_BEGIN;
        sound_end_address <= VICTORY_END;
        sample_hold_counter <= 3'd0;
    end

    SOUND_WALL_BREAK: begin
        state           <= PLAY_SFX;
        sound_address     <= WALL_BEGIN;
        sound_end_address <= WALL_END;
        sample_hold_counter <= 3'd0;
    end

    default: begin
        // SOUND_NONE or invalid Sound_ID:
```

```
            // keep current playback
        end
    endcase
end

if (audio_ready) begin
    sample_valid_l <= 1'b1;
    sample_valid_r <= 1'b1;
end

if (audio_ready_pulse) begin
    if (start_sfx_write) begin
        sample_hold_counter <= 3'd0;
    end else if (sample_hold_counter < 3'd5) begin
        sample_hold_counter <= sample_hold_counter + 3'd1;
    end else begin
        sample_hold_counter <= 3'd0;

        case (state)
            PLAY_BGM: begin
                if (sound_address >= `BGM_END) begin
                    sound_address <= `BGM_BEGIN;
                    bgm_address   <= `BGM_BEGIN;
                end else begin
                    sound_address <= sound_address + 18'd1;
                    bgm_address   <= bgm_address + 18'd1;
                end
            end

            PLAY_SFX: begin
                if (sound_address >= sound_end_address) begin
```

```
                    state            <= PLAY_BGM;
                    sound_address    <= bgm_address;
                    sound_end_address <= `BGM_END;
                end else begin
                    sound_address <= sound_address + 18'd1;
                end
            end

            default: begin
                state        <= PLAY_BGM;
                sound_address <= bgm_address;
            end
        endcase
    end
  end
 end

endmodule
```

8.2 Software

- Game

(1) game_logic.c

```
#include "game.h"
#include <string.h>

#define RESPAWN_STATE_NONE 255u
#define RESPAWN_STATE_SAFE_WAIT 254u
```

```
static const tile_id_t initial_map[MAP_ROWS][MAP_COLS] = {
    {TILE_HARD_WALL, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_HARD_WALL, TILE_HARD_WALL},
    {TILE_HARD_WALL, TILE_BLUE, TILE_HARD_WALL, TILE_BLUE,
TILE_BLUE, TILE_HARD_WALL, TILE_EMPTY, TILE_EMPTY,
TILE_EMPTY, TILE_EMPTY, TILE_EMPTY, TILE_EMPTY, TILE_EMPTY,
TILE_EMPTY, TILE_HARD_WALL, TILE_BLUE, TILE_BLUE,
TILE_HARD_WALL, TILE_BLUE, TILE_HARD_WALL},
    {TILE_HARD_WALL, TILE_BLUE, TILE_SOFT_WALL, TILE_BLUE,
TILE_BLUE, TILE_HARD_WALL, TILE_EMPTY, TILE_EMPTY,
TILE_EMPTY, TILE_EMPTY, TILE_EMPTY, TILE_EMPTY, TILE_EMPTY,
TILE_EMPTY, TILE_HARD_WALL, TILE_BLUE, TILE_BLUE,
TILE_HARD_WALL, TILE_BLUE, TILE_HARD_WALL},
    {TILE_HARD_WALL, TILE_BLUE, TILE_HARD_WALL, TILE_BLUE,
TILE_BLUE, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_HARD_WALL, TILE_BLUE, TILE_BLUE,
TILE_SOFT_WALL, TILE_BLUE, TILE_HARD_WALL},
    {TILE_HARD_WALL, TILE_BLUE, TILE_SOFT_WALL,
TILE_SOFT_WALL, TILE_BLUE, TILE_BLUE, TILE_BLUE, TILE_BLUE,
TILE_BLUE, TILE_HARD_WALL, TILE_BLUE, TILE_BLUE, TILE_BLUE,
```

TILE_BLUE, TILE_BLUE, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_BLUE, TILE_HARD_WALL},
{TILE_HARD_WALL, TILE_BLUE, TILE_BLUE, TILE_HARD_WALL,
TILE_BLUE, TILE_BLUE, TILE_BLUE, TILE_BLUE, TILE_BLUE,
TILE_SOFT_WALL, TILE_BLUE, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_BLUE, TILE_SOFT_WALL, TILE_BLUE,
TILE_BLUE, TILE_BLUE, TILE_HARD_WALL},
{TILE_HARD_WALL, TILE_BLUE, TILE_BLUE, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_SOFT_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_BLUE, TILE_BLUE, TILE_BLUE, TILE_BLUE, TILE_BLUE,
TILE_SOFT_WALL, TILE_BLUE, TILE_HARD_WALL, TILE_BLUE,
TILE_HARD_WALL},
{TILE_HARD_WALL, TILE_BLUE, TILE_BLUE, TILE_SOFT_WALL,
TILE_BLUE, TILE_BLUE, TILE_BLUE, TILE_BLUE, TILE_BLUE,
TILE_SOFT_WALL, TILE_BLUE, TILE_BLUE, TILE_BLUE, TILE_BLUE,
TILE_BLUE, TILE_HARD_WALL, TILE_BLUE, TILE_HARD_WALL,
TILE_BLUE, TILE_HARD_WALL},
{TILE_HARD_WALL, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_HARD_WALL, TILE_SOFT_WALL,
TILE_HARD_WALL, TILE_BLUE, TILE_BLUE, TILE_SOFT_WALL,
TILE_BLUE, TILE_HARD_WALL, TILE_SOFT_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_HARD_WALL, TILE_SOFT_WALL,
TILE_HARD_WALL, TILE_SOFT_WALL, TILE_HARD_WALL},
{TILE_HARD_WALL, TILE_BLUE, TILE_BLUE, TILE_BLUE,
TILE_BLUE, TILE_BLUE, TILE_HARD_WALL, TILE_BLUE, TILE_BLUE,
TILE_HARD_WALL, TILE_BLUE, TILE_BLUE, TILE_HARD_WALL,

TILE_BLUE, TILE_BLUE, TILE_BLUE, TILE_BLUE, TILE_HARD_WALL,
TILE_SOFT_WALL, TILE_HARD_WALL},
{TILE_HARD_WALL, TILE_HARD_WALL, TILE_SOFT_WALL,
TILE_HARD_WALL, TILE_BLUE, TILE_BLUE, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_SOFT_WALL, TILE_HARD_WALL, TILE_BLUE,
TILE_BLUE, TILE_SOFT_WALL, TILE_BLUE, TILE_BLUE, TILE_BLUE,
TILE_BLUE, TILE_HARD_WALL, TILE_BLUE, TILE_HARD_WALL},
{TILE_HARD_WALL, TILE_BLUE, TILE_BLUE, TILE_HARD_WALL,
TILE_BLUE, TILE_BLUE, TILE_SOFT_WALL, TILE_BLUE, TILE_BLUE,
TILE_BLUE, TILE_BLUE, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_BLUE, TILE_HARD_WALL, TILE_HARD_WALL, TILE_SOFT_WALL,
TILE_HARD_WALL, TILE_BLUE, TILE_HARD_WALL},
{TILE_HARD_WALL, TILE_BLUE, TILE_BLUE, TILE_SOFT_WALL,
TILE_BLUE, TILE_BLUE, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_BLUE, TILE_BLUE, TILE_SOFT_WALL,
TILE_BLUE, TILE_BLUE, TILE_BLUE, TILE_HARD_WALL, TILE_BLUE,
TILE_BLUE, TILE_BLUE, TILE_HARD_WALL},
{TILE_HARD_WALL, TILE_BLUE, TILE_BLUE, TILE_HARD_WALL,
TILE_BLUE, TILE_BLUE, TILE_BLUE, TILE_BLUE, TILE_HARD_WALL,
TILE_BLUE, TILE_BLUE, TILE_HARD_WALL, TILE_BLUE, TILE_BLUE,
TILE_BLUE, TILE_SOFT_WALL, TILE_BLUE, TILE_BLUE, TILE_BLUE,
TILE_HARD_WALL},
{TILE_HARD_WALL, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_HARD_WALL, TILE_HARD_WALL,

```
TILE_HARD_WALL, TILE_HARD_WALL, TILE_HARD_WALL,
TILE_HARD_WALL, TILE_HARD_WALL}
};

static controller_state_t prev_inputs[PLAYER_COUNT];
static uint8_t pending_soft_wall_hits[MAP_ROWS][MAP_COLS];
static uint8_t
explosion_marked_walls[ACTIVE_EXPLOSION_COUNT][EXPLOSION_TILE_
COUNT];
static uint8_t respawn_wait_state[PLAYER_COUNT];

static const uint8_t spawn_rows[PLAYER_COUNT] = {1, 1};
static const uint8_t spawn_cols[PLAYER_COUNT] = {1, 18};

static tile_id_t player_tile_id(uint8_t player_id)
{
   return player_id == 0 ? TILE_YELLOW : TILE_PINK;
}

static direction_t read_direction(const controller_state_t *input)
{
   if (input->up) {
      return DIR_UP;
   }
   if (input->down) {
      return DIR_DOWN;
   }
```

```
    if (input->left) {
        return DIR_LEFT;
    }
    if (input->right) {
        return DIR_RIGHT;
    }
    return DIR_NONE;
}

static uint8_t tile_is_enterable(tile_id_t tile)
{
    return tile == TILE_BLUE || tile == TILE_YELLOW || tile == TILE_PINK;
}

static void direction_delta(direction_t dir, int *dr, int *dc)
{
    *dr = 0;
    *dc = 0;
    if (dir == DIR_UP) {
        *dr = -1;
    } else if (dir == DIR_DOWN) {
        *dr = 1;
    } else if (dir == DIR_LEFT) {
        *dc = -1;
    } else if (dir == DIR_RIGHT) {
        *dc = 1;
    }
```

```
}

static int find_active_bomb_at(const game_state_t *game, uint8_t row, uint8_t col)
{
    int i;
    for (i = 0; i < ACTIVE_BOMB_COUNT; ++i) {
        if (game->bombs[i].active && game->bombs[i].row == row &&
game->bombs[i].col == col) {
            return i;
        }
    }
    return -1;
}

static uint8_t tile_blocked_by_other_player(const game_state_t *game, uint8_t
player_index, uint8_t row, uint8_t col)
{
    const player_t *other = &game->players[1 - player_index];

    if (!other->alive) {
        return 0;
    }
    if (other->row == row && other->col == col) {
        return 1;
    }
    if (other->moving && other->target_row == row && other->target_col == col)
{
```

```
        return 1;
    }
    return 0;
}

static uint8_t can_enter_tile(const game_state_t *game, uint8_t player_index,
uint8_t row, uint8_t col)
{
    if (row >= MAP_ROWS || col >= MAP_COLS) {
        return 0;
    }
    if (!tile_is_enterable(game->tiles[row][col])) {
        return 0;
    }
    if (find_active_bomb_at(game, row, col) >= 0) {
        return 0;
    }
    if (tile_blocked_by_other_player(game, player_index, row, col)) {
        return 0;
    }
    return 1;
}

static void set_player_target(player_t *player, direction_t dir, uint8_t row, uint8_t
col)
{
    player->moving = 1;
```

```
    player->dir = dir;
    player->target_row = row;
    player->target_col = col;
    player->target_x = (uint16_t)(col * TILE_SIZE);
    player->target_y = (uint16_t)(row * TILE_SIZE);
}

static uint8_t step_toward_u16(uint16_t *value, uint16_t target)
{
    if (*value < target) {
        uint16_t delta = (uint16_t)(target - *value);
        *value = (uint16_t)(*value + (delta > PLAYER_SPEED ?
PLAYER_SPEED : delta));
    } else if (*value > target) {
        uint16_t delta = (uint16_t)(*value - target);
        *value = (uint16_t)(*value - (delta > PLAYER_SPEED ? PLAYER_SPEED :
delta));
    }
    return *value == target;
}

static uint8_t advance_player_motion(player_t *player)
{
    uint8_t reached_x;
    uint8_t reached_y;

    if (!player->moving) {
```

```
        return 0;
    }

    reached_x = step_toward_u16(&player->x, player->target_x);
    reached_y = step_toward_u16(&player->y, player->target_y);

    if (reached_x && reached_y) {
        player->x = player->target_x;
        player->y = player->target_y;
        player->row = player->target_row;
        player->col = player->target_col;
        player->moving = 0;
        return 1;
    }

    return 0;
}

static void apply_tile_capture(game_state_t *game, uint8_t player_index)
{
    player_t *player = &game->players[player_index];
    tile_id_t own_tile = player_tile_id(player_index);
    tile_id_t opponent_tile = player_tile_id((uint8_t)(1 - player_index));
    tile_id_t *tile = &game->tiles[player->row][player->col];

    if (*tile == TILE_BLUE) {
        *tile = own_tile;
```

```
        player->score = (uint16_t)(player->score + 1);
    } else if (*tile == opponent_tile) {
        *tile = own_tile;
        player->score = (uint16_t)(player->score + 1);
        if (game->players[1 - player_index].score > 0) {
            game->players[1 - player_index].score = (uint16_t)(game->players[1 -
player_index].score - 1);
        }
    }
}

static uint8_t build_move_candidate(const game_state_t *game, uint8_t
player_index, direction_t dir, uint8_t *target_row, uint8_t *target_col)
{
    const player_t *player = &game->players[player_index];
    int dr;
    int dc;
    int next_row;
    int next_col;

    if (!player->alive || player->moving || dir == DIR_NONE) {
        return 0;
    }

    direction_delta(dir, &dr, &dc);
    next_row = (int)player->row + dr;
    next_col = (int)player->col + dc;
```

```
    if (next_row < 0 || next_row >= MAP_ROWS || next_col < 0 || next_col >=
MAP_COLS) {
        return 0;
    }

    *target_row = (uint8_t)next_row;
    *target_col = (uint8_t)next_col;

    return can_enter_tile(game, player_index, *target_row, *target_col);
}

static void clear_player_territory(game_state_t *game, uint8_t player_index)
{
    tile_id_t own_tile = player_tile_id(player_index);
    int row;
    int col;

    for (row = 0; row < MAP_ROWS; ++row) {
        for (col = 0; col < MAP_COLS; ++col) {
            if (game->tiles[row][col] == own_tile) {
                game->tiles[row][col] = TILE_BLUE;
            }
        }
    }
}
```

```
static uint8_t rect_overlaps_tile(uint16_t x, uint16_t y, uint8_t row, uint8_t col)
{
    int ax1 = x;
    int ay1 = y;
    int ax2 = ax1 + TILE_SIZE - 1;
    int ay2 = ay1 + TILE_SIZE - 1;
    int bx1 = col * TILE_SIZE;
    int by1 = row * TILE_SIZE;
    int bx2 = bx1 + TILE_SIZE - 1;
    int by2 = by1 + TILE_SIZE - 1;

    if (ax2 < bx1 || bx2 < ax1) {
        return 0;
    }
    if (ay2 < by1 || by2 < ay1) {
        return 0;
    }
    return 1;
}

static uint8_t explosion_hits_player(const explosion_t *explosion, const player_t
*player)
{
    int i;
    for (i = 0; i < EXPLOSION_TILE_COUNT; ++i) {
        if (rect_overlaps_tile(player->x, player->y, explosion->cells[i].row,
explosion->cells[i].col)) {
```

```
            return 1;
        }
    }
    return 0;
}

static void kill_player(game_state_t *game, uint8_t player_index, uint8_t
explosion_index)
{
    player_t *player = &game->players[player_index];

    if (!player->alive) {
        return;
    }

    player->alive = 0;
    player->moving = 0;
    player->pending_dir = DIR_NONE;
    player->target_row = player->row;
    player->target_col = player->col;
    player->target_x = player->x;
    player->target_y = player->y;

    clear_player_territory(game, player_index);
    player->score = 0;

    respawn_wait_state[player_index] = explosion_index;
```

```
}

static void fill_explosion_cells(explosion_t *explosion)
{
    explosion->cells[0].row = explosion->center_row;
    explosion->cells[0].col = explosion->center_col;
    explosion->cells[1].row = (uint8_t)(explosion->center_row - 1);
    explosion->cells[1].col = explosion->center_col;
    explosion->cells[2].row = (uint8_t)(explosion->center_row + 1);
    explosion->cells[2].col = explosion->center_col;
    explosion->cells[3].row = explosion->center_row;
    explosion->cells[3].col = (uint8_t)(explosion->center_col - 1);
    explosion->cells[4].row = explosion->center_row;
    explosion->cells[4].col = (uint8_t)(explosion->center_col + 1);
}

static uint8_t create_explosion_from_bomb(game_state_t *game, uint8_t index)
{
    bomb_t *bomb = &game->bombs[index];
    explosion_t *explosion = &game->explosions[index];
    uint8_t hit_soft_wall = 0;
    int i;

    explosion->active = 1;
    explosion->owner = bomb->owner;
    explosion->center_row = bomb->row;
    explosion->center_col = bomb->col;
```

```
    explosion->timer = EXPLOSION_TIMER_TICKS;
    fill_explosion_cells(explosion);

    memset(explosion_marked_walls[index], 0,
sizeof(explosion_marked_walls[index]));

    for (i = 0; i < EXPLOSION_TILE_COUNT; ++i) {
        uint8_t row = explosion->cells[i].row;
        uint8_t col = explosion->cells[i].col;
        if (game->tiles[row][col] == TILE_SOFT_WALL) {
            pending_soft_wall_hits[row][col] =
(uint8_t)(pending_soft_wall_hits[row][col] + 1);
            explosion_marked_walls[index][i] = 1;
            hit_soft_wall = 1;
        }
    }

    bomb->active = 0;
    bomb->timer = 0;

    return hit_soft_wall;
}

static void cleanup_explosion(game_state_t *game, uint8_t index)
{
    explosion_t *explosion = &game->explosions[index];
    int i;
```

```
    for (i = 0; i < EXPLOSION_TILE_COUNT; ++i) {
        uint8_t row = explosion->cells[i].row;
        uint8_t col = explosion->cells[i].col;

        if (explosion_marked_walls[index][i]) {
            if (pending_soft_wall_hits[row][col] > 0) {
                pending_soft_wall_hits[row][col] =
(uint8_t)(pending_soft_wall_hits[row][col] - 1);
            }
            if (pending_soft_wall_hits[row][col] == 0 && game->tiles[row][col] ==
TILE_SOFT_WALL) {
                game->tiles[row][col] = TILE_BLUE;
            }
            explosion_marked_walls[index][i] = 0;
        }
    }

    explosion->active = 0;
    explosion->timer = 0;
}

static uint8_t tile_in_active_explosion(const game_state_t *game, uint8_t row,
uint8_t col)
{
    int i;
    int j;
```

```
    for (i = 0; i < ACTIVE_EXPLOSION_COUNT; ++i) {
        if (!game->explosions[i].active) {
            continue;
        }
        for (j = 0; j < EXPLOSION_TILE_COUNT; ++j) {
            if (game->explosions[i].cells[j].row == row &&
game->explosions[i].cells[j].col == col) {
                return 1;
            }
        }
    }

    return 0;
}

static uint8_t spawn_tile_safe(const game_state_t *game, uint8_t player_index)
{
    uint8_t row = spawn_rows[player_index];
    uint8_t col = spawn_cols[player_index];
    const player_t *other = &game->players[1 - player_index];

    if (!tile_is_enterable(game->tiles[row][col])) {
        return 0;
    }
    if (find_active_bomb_at(game, row, col) >= 0) {
        return 0;
```

```
    }
    if (tile_in_active_explosion(game, row, col)) {
        return 0;
    }
    if (other->alive) {
        if (other->row == row && other->col == col) {
            return 0;
        }
        if (other->moving && other->target_row == row && other->target_col ==
col) {
            return 0;
        }
    }
    return 1;
}

static void respawn_player(game_state_t *game, uint8_t player_index)
{
    player_t *player = &game->players[player_index];
    tile_id_t own_tile = player_tile_id(player_index);
    tile_id_t opponent_tile = player_tile_id((uint8_t)(1 - player_index));
    tile_id_t *spawn_tile;

    player->alive = 1;
    player->moving = 0;
    player->row = spawn_rows[player_index];
    player->col = spawn_cols[player_index];
```

```
    player->x = (uint16_t)(player->col * TILE_SIZE);
    player->y = (uint16_t)(player->row * TILE_SIZE);
    player->target_row = player->row;
    player->target_col = player->col;
    player->target_x = player->x;
    player->target_y = player->y;
    player->dir = DIR_DOWN;
    player->pending_dir = DIR_NONE;

    spawn_tile = &game->tiles[player->row][player->col];

    if (*spawn_tile == TILE_BLUE) {
        *spawn_tile = own_tile;
        player->score = (uint16_t)(player->score + 1);
    } else if (*spawn_tile == opponent_tile) {
        *spawn_tile = own_tile;
        player->score = (uint16_t)(player->score + 1);
        if (game->players[1 - player_index].score > 0) {
            game->players[1 - player_index].score = (uint16_t)(game->players[1 -
player_index].score - 1);
        }
    } else if (*spawn_tile == own_tile && player->score == 0) {
        player->score = 1;
    }

    respawn_wait_state[player_index] = RESPAWN_STATE_NONE;
}
```

```
static void reset_internal_state(void)
{
    memset(prev_inputs, 0, sizeof(prev_inputs));
    memset(pending_soft_wall_hits, 0, sizeof(pending_soft_wall_hits));
    memset(explosion_marked_walls, 0, sizeof(explosion_marked_walls));
    memset(respawn_wait_state, RESPAWN_STATE_NONE,
sizeof(respawn_wait_state));
}

void game_reset(game_state_t *game)
{
    int i;

    memcpy(game->tiles, initial_map, sizeof(initial_map));
    memset(game->bombs, 0, sizeof(game->bombs));
    memset(game->explosions, 0, sizeof(game->explosions));

    for (i = 0; i < PLAYER_COUNT; ++i) {
        game->players[i].id = (uint8_t)i;
        game->players[i].alive = 1;
        game->players[i].row = spawn_rows[i];
        game->players[i].col = spawn_cols[i];
        game->players[i].x = (uint16_t)(spawn_cols[i] * TILE_SIZE);
        game->players[i].y = (uint16_t)(spawn_rows[i] * TILE_SIZE);
        game->players[i].target_row = game->players[i].row;
        game->players[i].target_col = game->players[i].col;
```

```
    game->players[i].target_x = game->players[i].x;
    game->players[i].target_y = game->players[i].y;
    game->players[i].moving = 0;
    game->players[i].score = 1;
    game->players[i].dir = DIR_DOWN;
    game->players[i].pending_dir = DIR_NONE;

    game->bombs[i].active = 0;
    game->bombs[i].owner = (uint8_t)i;
    game->bombs[i].row = spawn_rows[i];
    game->bombs[i].col = spawn_cols[i];
    game->bombs[i].timer = 0;

    game->explosions[i].active = 0;
    game->explosions[i].owner = (uint8_t)i;
    game->explosions[i].center_row = spawn_rows[i];
    game->explosions[i].center_col = spawn_cols[i];
    game->explosions[i].timer = 0;
    memset(game->explosions[i].cells, 0, sizeof(game->explosions[i].cells));

    game->tiles[spawn_rows[i]][spawn_cols[i]] = player_tile_id((uint8_t)i);
}

game->pending_sound_1 = SOUND_NONE;
game->pending_sound_2 = SOUND_NONE;
game->game_over = 0;
game->winner = -1;
```

```
    reset_internal_state();
}

void game_init(game_state_t *game)
{
    game_reset(game);
}

void game_step(game_state_t *game, const controller_state_t
inputs[PLAYER_COUNT])
{
    direction_t desired_dirs[PLAYER_COUNT];
    uint8_t arrived[PLAYER_COUNT] = {0, 0};
    uint8_t candidate_valid[PLAYER_COUNT] = {0, 0};
    uint8_t candidate_row[PLAYER_COUNT] = {0, 0};
    uint8_t candidate_col[PLAYER_COUNT] = {0, 0};
    uint8_t explosion_triggered = 0;
    uint8_t wall_hit_this_frame = 0;
    uint8_t player_died_this_frame = 0;
    int i;

    if (inputs[0].restart || inputs[1].restart) {
        game_reset(game);
        return;
    }
```

```
game->pending_sound_1 = SOUND_NONE;
game->pending_sound_2 = SOUND_NONE;

if (game->game_over) {
   for (i = 0; i < PLAYER_COUNT; ++i) {
      prev_inputs[i] = inputs[i];
   }
   return;
}

for (i = 0; i < PLAYER_COUNT; ++i) {
   desired_dirs[i] = read_direction(&inputs[i]);
   if (game->players[i].alive) {
      game->players[i].pending_dir = desired_dirs[i];
   } else {
      game->players[i].pending_dir = DIR_NONE;
   }
}

for (i = 0; i < PLAYER_COUNT; ++i) {
   if (game->players[i].alive) {
      arrived[i] = advance_player_motion(&game->players[i]);
   }
}

for (i = 0; i < PLAYER_COUNT; ++i) {
   if (arrived[i] && game->players[i].alive) {
```

```
            apply_tile_capture(game, (uint8_t)i);
        }
    }

    for (i = 0; i < PLAYER_COUNT; ++i) {
        if (build_move_candidate(game, (uint8_t)i, game->players[i].pending_dir,
&candidate_row[i], &candidate_col[i])) {
            candidate_valid[i] = 1;
        }
    }

    if (candidate_valid[0] && candidate_valid[1] &&
        candidate_row[0] == candidate_row[1] &&
        candidate_col[0] == candidate_col[1]) {
        candidate_valid[0] = 0;
        candidate_valid[1] = 0;
    }

    for (i = 0; i < PLAYER_COUNT; ++i) {
        if (candidate_valid[i]) {
            set_player_target(&game->players[i], game->players[i].pending_dir,
candidate_row[i], candidate_col[i]);
        }
    }

    for (i = 0; i < PLAYER_COUNT; ++i) {
        uint8_t bomb_pressed = (uint8_t)(inputs[i].bomb && !prev_inputs[i].bomb);
```

```
    player_t *player = &game->players[i];
    bomb_t *bomb = &game->bombs[i];
    explosion_t *explosion = &game->explosions[i];

    if (!bomb_pressed) {
        continue;
    }
    if (!player->alive || player->moving) {
        continue;
    }
    if (bomb->active || explosion->active) {
        continue;
    }
    if (!tile_is_enterable(game->tiles[player->row][player->col])) {
        continue;
    }
    if (find_active_bomb_at(game, player->row, player->col) >= 0) {
        continue;
    }

    bomb->active = 1;
    bomb->owner = (uint8_t)i;
    bomb->row = player->row;
    bomb->col = player->col;
    bomb->timer = BOMB_TIMER_TICKS;
}
```

```
    for (i = 0; i < ACTIVE_BOMB_COUNT; ++i) {
        if (!game->bombs[i].active) {
            continue;
        }
        if (game->bombs[i].timer > 0) {
            game->bombs[i].timer--;
        }
        if (game->bombs[i].timer == 0) {
            explosion_triggered = 1;
            if (create_explosion_from_bomb(game, (uint8_t)i)) {
                wall_hit_this_frame = 1;
            }
        }
    }

    for (i = 0; i < ACTIVE_EXPLOSION_COUNT; ++i) {
        int p;
        if (!game->explosions[i].active) {
            continue;
        }
        for (p = 0; p < PLAYER_COUNT; ++p) {
            if (game->players[p].alive &&
explosion_hits_player(&game->explosions[i], &game->players[p])) {
                kill_player(game, (uint8_t)p, (uint8_t)i);
                player_died_this_frame = 1;
            }
        }
```

```
}

for (i = 0; i < ACTIVE_EXPLOSION_COUNT; ++i) {
    if (!game->explosions[i].active) {
        continue;
    }
    if (game->explosions[i].timer > 0) {
        game->explosions[i].timer--;
    }
    if (game->explosions[i].timer == 0) {
        cleanup_explosion(game, (uint8_t)i);
    }
}

for (i = 0; i < PLAYER_COUNT; ++i) {
    if (game->players[i].alive) {
        continue;
    }

    if (respawn_wait_state[i] < ACTIVE_EXPLOSION_COUNT) {
        if (game->explosions[respawn_wait_state[i]].active) {
            continue;
        }
        respawn_wait_state[i] = RESPAWN_STATE_SAFE_WAIT;
    }
```

```
        if (respawn_wait_state[i] == RESPAWN_STATE_SAFE_WAIT &&
spawn_tile_safe(game, (uint8_t)i)) {
            respawn_player(game, (uint8_t)i);
        }
    }

    if (game->players[0].score >= WIN_SCORE && game->players[1].score >=
WIN_SCORE) {
        game->game_over = 1;
        if (game->players[0].score > game->players[1].score) {
            game->winner = 0;
        } else if (game->players[1].score > game->players[0].score) {
            game->winner = 1;
        } else {
            game->winner = -1;
        }
    } else if (game->players[0].score >= WIN_SCORE) {
        game->game_over = 1;
        game->winner = 0;
    } else if (game->players[1].score >= WIN_SCORE) {
        game->game_over = 1;
        game->winner = 1;
    }

    if (game->game_over) {
        game->pending_sound_1 = SOUND_VICTORY;
        game->pending_sound_2 = SOUND_NONE;
```

```
    } else if (explosion_triggered) {
        game->pending_sound_1 = SOUND_EXPLOSION;
        if (player_died_this_frame) {
            game->pending_sound_2 = SOUND_PLAYER_DIES;
        } else if (wall_hit_this_frame) {
            game->pending_sound_2 = SOUND_WALL_BREAKS;
        }
    } else if (player_died_this_frame) {
        game->pending_sound_1 = SOUND_PLAYER_DIES;
    }

    for (i = 0; i < PLAYER_COUNT; ++i) {
        prev_inputs[i] = inputs[i];
    }
}
```

(2) game.h

```
#ifndef GAME_H
#define GAME_H

#include <stdint.h>

#define MAP_ROWS 15
#define MAP_COLS 20
#define TILE_SIZE 32
#define SCREEN_WIDTH 640
#define SCREEN_HEIGHT 480
```

```
#define PLAYER_COUNT 2
#define ACTIVE_BOMB_COUNT 2
#define ACTIVE_EXPLOSION_COUNT 2
#define EXPLOSION_TILE_COUNT 5
#define SPRITE_REGISTER_COUNT 30
#define SPRITE_ID_COUNT 30
#define TILE_ID_COUNT 6
#define WIN_SCORE 50

#define GAME_TICK_HZ 32
#define PLAYER_SPEED 2
#define BOMB_TIMER_TICKS 53
#define EXPLOSION_TIMER_TICKS 32

#define UI_ROW_START 1
#define UI_ROW_END 2
#define UI_COL_START 6
#define UI_COL_END 13

typedef enum {
    TILE_EMPTY = 0,
    TILE_BLUE = 1,
    TILE_PINK = 2,
    TILE_YELLOW = 3,
    TILE_HARD_WALL = 4,
    TILE_SOFT_WALL = 5
} tile_id_t;
```

```
typedef enum {
    DIR_UP = 0,
    DIR_DOWN = 1,
    DIR_LEFT = 2,
    DIR_RIGHT = 3,
    DIR_NONE = 4
} direction_t;

typedef enum {
    SPRITE_PLAYER1_FRONT = 0,
    SPRITE_PLAYER1_BACK = 1,
    SPRITE_PLAYER1_LEFT = 2,
    SPRITE_PLAYER1_RIGHT = 3,
    SPRITE_PLAYER2_FRONT = 4,
    SPRITE_PLAYER2_BACK = 5,
    SPRITE_PLAYER2_LEFT = 6,
    SPRITE_PLAYER2_RIGHT = 7,
    SPRITE_PLAYER1_BOMB = 8,
    SPRITE_PLAYER2_BOMB = 9,
    SPRITE_PLAYER1_EXPLOSION_CENTER = 10,
    SPRITE_PLAYER2_EXPLOSION_CENTER = 11,
    SPRITE_PLAYER1_EXPLOSION_UP = 12,
    SPRITE_PLAYER1_EXPLOSION_DOWN = 13,
    SPRITE_PLAYER2_EXPLOSION_UP = 14,
    SPRITE_PLAYER2_EXPLOSION_DOWN = 15,
    SPRITE_PLAYER1_EXPLOSION_LEFT = 16,
```

```
    SPRITE_PLAYER1_EXPLOSION_RIGHT = 17,
    SPRITE_PLAYER2_EXPLOSION_LEFT = 18,
    SPRITE_PLAYER2_EXPLOSION_RIGHT = 19,
    SPRITE_UI_CHARACTER_S = 20,
    SPRITE_UI_CHARACTER_V = 21,
    SPRITE_UI_PINK_W = 22,
    SPRITE_UI_PINK_I = 23,
    SPRITE_UI_PINK_N = 24,
    SPRITE_UI_YELLOW_W = 25,
    SPRITE_UI_YELLOW_I = 26,
    SPRITE_UI_YELLOW_N = 27,
    SPRITE_UI_PLAYER1 = 28,
    SPRITE_UI_PLAYER2 = 29
} sprite_id_t;

typedef enum {
    SOUND_NONE = 0,
    SOUND_EXPLOSION = 1,
    SOUND_PLAYER_DIES = 2,
    SOUND_VICTORY = 3,
    SOUND_WALL_BREAKS = 4
} sound_id_t;

typedef struct {
    uint8_t up;
    uint8_t down;
    uint8_t left;
```

```
    uint8_t right;
    uint8_t bomb;
    uint8_t restart;
} controller_state_t;

typedef struct {
    uint8_t row;
    uint8_t col;
} grid_pos_t;

typedef struct {
    uint8_t active;
    uint8_t owner;
    uint8_t row;
    uint8_t col;
    uint32_t timer;
} bomb_t;

typedef struct {
    uint8_t active;
    uint8_t owner;
    uint8_t center_row;
    uint8_t center_col;
    uint32_t timer;
    grid_pos_t cells[EXPLOSION_TILE_COUNT];
} explosion_t;
```

```
typedef struct {
    uint8_t id;
    uint8_t alive;
    uint8_t row;
    uint8_t col;
    uint16_t x;
    uint16_t y;
    uint8_t target_row;
    uint8_t target_col;
    uint16_t target_x;
    uint16_t target_y;
    uint8_t moving;
    uint16_t score;
    direction_t dir;
    direction_t pending_dir;
} player_t;

typedef struct {
    tile_id_t tiles[MAP_ROWS][MAP_COLS];
    player_t players[PLAYER_COUNT];
    bomb_t bombs[ACTIVE_BOMB_COUNT];
    explosion_t explosions[ACTIVE_EXPLOSION_COUNT];
    sound_id_t pending_sound_1;
    sound_id_t pending_sound_2;
    uint8_t game_over;
    int8_t winner;
} game_state_t;
```

```
typedef struct {
    uint8_t row;
    uint8_t col;
    tile_id_t tile_id;
} tile_t;

typedef struct {
    uint8_t sprite_index;
    uint16_t x;
    uint16_t y;
    sprite_id_t sprite_id;
    uint8_t enable;
} sprite_t;

void game_init(game_state_t *game);
void game_reset(game_state_t *game);
void game_step(game_state_t *game, const controller_state_t
inputs[PLAYER_COUNT]);

#endif
```

(3) usbcontroller.c

```
#include "usbcontroller.h"

#include <libusb-1.0/libusb.h>
#include <string.h>
#include <sys/types.h>
```

```
#define USB_VENDOR_ID 0x0079
#define USB_PRODUCT_ID 0x0011
#define USB_INTERFACE_NUMBER 0
#define USB_ENDPOINT_IN 0x81
#define USB_PACKET_SIZE 8
#define USB_READ_TIMEOUT_MS 5

static libusb_context *g_usb_ctx = NULL;
static libusb_device_handle *g_handles[PLAYER_COUNT];
static uint8_t g_detached[PLAYER_COUNT];
static controller_state_t g_cached_inputs[PLAYER_COUNT];
static uint8_t g_usb_initialized = 0;

static void clear_controller_state(controller_state_t *state)
{
   memset(state, 0, sizeof(*state));
}

static void parse_packet(const unsigned char packet[USB_PACKET_SIZE],
controller_state_t *state)
{
   state->left = (packet[3] == 0x00);
   state->right = (packet[3] == 0xff);
   state->up = (packet[4] == 0x00);
   state->down = (packet[4] == 0xff);
   state->bomb = ((packet[5] & 0x20) != 0);
```

```
    state->restart = ((packet[6] & 0x20) != 0);
}

static void close_one_handle(int index)
{
    if (index < 0 || index >= PLAYER_COUNT) {
        return;
    }

    if (g_handles[index] != NULL) {
        libusb_release_interface(g_handles[index], USB_INTERFACE_NUMBER);
        if (g_detached[index]) {
            libusb_attach_kernel_driver(g_handles[index],
USB_INTERFACE_NUMBER);
        }
        libusb_close(g_handles[index]);
    }

    g_handles[index] = NULL;
    g_detached[index] = 0;
    clear_controller_state(&g_cached_inputs[index]);
}

static int open_matching_devices(void)
{
    libusb_device **list = NULL;
    ssize_t count;
```

```
    ssize_t i;
    int opened = 0;

    count = libusb_get_device_list(g_usb_ctx, &list);
    if (count < 0) {
        return -1;
    }

    for (i = 0; i < count && opened < PLAYER_COUNT; ++i) {
        struct libusb_device_descriptor desc;
        libusb_device_handle *handle = NULL;
        int ret;
        int active;

        ret = libusb_get_device_descriptor(list[i], &desc);
        if (ret != 0) {
            continue;
        }

        if (desc.idVendor != USB_VENDOR_ID || desc.idProduct !=
USB_PRODUCT_ID) {
            continue;
        }

        ret = libusb_open(list[i], &handle);
        if (ret != 0 || handle == NULL) {
            continue;
```

```
        }

        active = libusb_kernel_driver_active(handle,
USB_INTERFACE_NUMBER);
        if (active == 1) {
            ret = libusb_detach_kernel_driver(handle, USB_INTERFACE_NUMBER);
            if (ret != 0) {
                libusb_close(handle);
                continue;
            }
            g_detached[opened] = 1;
        } else {
            g_detached[opened] = 0;
        }

        ret = libusb_claim_interface(handle, USB_INTERFACE_NUMBER);
        if (ret != 0) {
            if (g_detached[opened]) {
                libusb_attach_kernel_driver(handle, USB_INTERFACE_NUMBER);
                g_detached[opened] = 0;
            }
            libusb_close(handle);
            continue;
        }

        g_handles[opened] = handle;
        clear_controller_state(&g_cached_inputs[opened]);
```

```
        opened++;
    }

    libusb_free_device_list(list, 1);

    return opened > 0 ? 0 : -1;
}

int usb_init(void)
{
    int i;
    int ret;

    if (g_usb_initialized) {
        return 0;
    }

    ret = libusb_init(&g_usb_ctx);
    if (ret != 0) {
        g_usb_ctx = NULL;
        return -1;
    }

    for (i = 0; i < PLAYER_COUNT; ++i) {
        g_handles[i] = NULL;
        g_detached[i] = 0;
        clear_controller_state(&g_cached_inputs[i]);
```

```
    }

    g_usb_initialized = 1;
    return 0;
}

int usb_open(void)
{
    int i;

    if (!g_usb_initialized) {
        if (usb_init() != 0) {
            return -1;
        }
    }

    for (i = 0; i < PLAYER_COUNT; ++i) {
        close_one_handle(i);
    }

    return open_matching_devices();
}

int usb_read(controller_state_t inputs[PLAYER_COUNT])
{
    int i;
```

```
if (!g_usb_initialized) {
   return -1;
}

for (i = 0; i < PLAYER_COUNT; ++i) {
   clear_controller_state(&inputs[i]);
}

for (i = 0; i < PLAYER_COUNT; ++i) {
   unsigned char packet[USB_PACKET_SIZE];
   int transferred = 0;
   int ret;

   if (g_handles[i] == NULL) {
      continue;
   }

   ret = libusb_interrupt_transfer(
      g_handles[i],
      USB_ENDPOINT_IN,
      packet,
      USB_PACKET_SIZE,
      &transferred,
      USB_READ_TIMEOUT_MS
   );

   if (ret == 0 && transferred == USB_PACKET_SIZE) {
```

```
            parse_packet(packet, &g_cached_inputs[i]);
        } else if (ret == LIBUSB_ERROR_TIMEOUT) {
        } else if (ret == LIBUSB_ERROR_NO_DEVICE) {
            close_one_handle(i);
        } else {
            close_one_handle(i);
        }

        inputs[i] = g_cached_inputs[i];
    }

    return 0;
}

void usb_close(void)
{
    int i;

    for (i = 0; i < PLAYER_COUNT; ++i) {
        close_one_handle(i);
    }

    if (g_usb_initialized) {
        libusb_exit(g_usb_ctx);
        g_usb_ctx = NULL;
        g_usb_initialized = 0;
    }
```

}

(4) usbcontroller.h

```
#ifndef USBCONTROLLER_H
#define USBCONTROLLER_H

#include "game.h"

int usb_init(void);
int usb_open(void);
int usb_read(controller_state_t inputs[PLAYER_COUNT]);
void usb_close(void);

#endif
```

(5) audio_interface.c

```
#define _POSIX_C_SOURCE 200809L

#include "audio_interface.h"
#include "../kernel_modules/audio/fpga_audio.h"

#include <errno.h>
#include <fcntl.h>
#include <stdint.h>
#include <time.h>
#include <unistd.h>

#define AUDIO_QUEUE_CAPACITY 16
```

```
#define SOUND_EXPLOSION_DURATION_MS 350u
#define SOUND_PLAYER_DIES_DURATION_MS 180u
#define SOUND_VICTORY_DURATION_MS 350u
#define SOUND_WALL_BREAKS_DURATION_MS 180u

static int audio_fd = -1;

static sound_id_t audio_queue[AUDIO_QUEUE_CAPACITY];
static uint8_t queue_head = 0;
static uint8_t queue_tail = 0;
static uint8_t queue_count = 0;

static uint8_t current_sound_active = 0;
static uint64_t current_sound_end_ms = 0;

static int sound_is_valid(sound_id_t sound)
{
   return sound >= SOUND_NONE && sound <= SOUND_WALL_BREAKS;
}

static uint32_t sound_duration_ms(sound_id_t sound)
{
   if (sound == SOUND_EXPLOSION) {
      return SOUND_EXPLOSION_DURATION_MS;
   }
   if (sound == SOUND_PLAYER_DIES) {
      return SOUND_PLAYER_DIES_DURATION_MS;
```

```
    }
    if (sound == SOUND_VICTORY) {
        return SOUND_VICTORY_DURATION_MS;
    }
    if (sound == SOUND_WALL_BREAKS) {
        return SOUND_WALL_BREAKS_DURATION_MS;
    }
    return 0;
}

static int monotonic_ms(uint64_t *out_ms)
{
    struct timespec ts;

    if (clock_gettime(CLOCK_MONOTONIC, &ts) != 0) {
        return -1;
    }

    *out_ms = (uint64_t)ts.tv_sec * 1000u + (uint64_t)ts.tv_nsec / 1000000u;
    return 0;
}

static void queue_clear_only(void)
{
    queue_head = 0;
    queue_tail = 0;
    queue_count = 0;
```

```
}

static int queue_push(sound_id_t sound)
{
    if (queue_count >= AUDIO_QUEUE_CAPACITY) {
        return 0;
    }

    audio_queue[queue_tail] = sound;
    queue_tail = (uint8_t)((queue_tail + 1) % AUDIO_QUEUE_CAPACITY);
    queue_count++;

    return 0;
}

static int queue_pop(sound_id_t *sound)
{
    if (queue_count == 0) {
        return 0;
    }

    *sound = audio_queue[queue_head];
    queue_head = (uint8_t)((queue_head + 1) % AUDIO_QUEUE_CAPACITY);
    queue_count--;

    return 1;
}
```

```
static int start_sound_now(sound_id_t sound)
{
    uint64_t now;
    uint32_t duration;

    if (sound == SOUND_NONE) {
        return 0;
    }

    if (!sound_is_valid(sound)) {
        errno = EINVAL;
        return -1;
    }

    if (monotonic_ms(&now) != 0) {
        return -1;
    }

    if (play_sound_id(sound) != 0) {
        return -1;
    }

    duration = sound_duration_ms(sound);
    if (duration == 0) {
        current_sound_active = 0;
        current_sound_end_ms = 0;
```

```
    } else {
        current_sound_active = 1;
        current_sound_end_ms = now + duration;
    }

    return 0;
}

int audio_init(void)
{
    if (audio_fd >= 0) {
        return 0;
    }

    audio_fd = open("/dev/fpga_audio", O_WRONLY);
    if (audio_fd < 0) {
        return -1;
    }

    queue_clear_only();
    current_sound_active = 0;
    current_sound_end_ms = 0;

    return 0;
}

void audio_close(void)
```

```
{
    if (audio_fd >= 0) {
        close(audio_fd);
        audio_fd = -1;
    }

    queue_clear_only();
    current_sound_active = 0;
    current_sound_end_ms = 0;
}

int play_sound_id(sound_id_t sound)
{
    fpga_audio_u32 id;
    ssize_t written;

    if (sound == SOUND_NONE) {
        return 0;
    }

    if (!sound_is_valid(sound)) {
        errno = EINVAL;
        return -1;
    }

    if (audio_fd < 0) {
        errno = ENODEV;
```

```
        return -1;
    }

    id = (fpga_audio_u32)sound;
    written = write(audio_fd, &id, sizeof(id));

    if (written == (ssize_t)sizeof(id)) {
        return 0;
    }

    if (written >= 0) {
        errno = EIO;
    }

    return -1;
}

int audio_enqueue_sound(sound_id_t sound)
{
    if (sound == SOUND_NONE) {
        return 0;
    }

    if (!sound_is_valid(sound)) {
        errno = EINVAL;
        return -1;
    }
```

```
    if (sound == SOUND_VICTORY) {
        queue_clear_only();
        return start_sound_now(SOUND_VICTORY);
    }

    return queue_push(sound);
}

int audio_submit_frame_sounds(sound_id_t sound1, sound_id_t sound2)
{
    if (sound1 == SOUND_VICTORY) {
        return audio_enqueue_sound(sound1);
    }

    if (sound2 == SOUND_VICTORY) {
        return audio_enqueue_sound(sound2);
    }

    if (audio_enqueue_sound(sound1) != 0) {
        return -1;
    }

    if (audio_enqueue_sound(sound2) != 0) {
        return -1;
    }
```

```
    return 0;
}

int audio_update(void)
{
    uint64_t now;
    sound_id_t next_sound;

    if (audio_fd < 0) {
        errno = ENODEV;
        return -1;
    }

    if (current_sound_active) {
        if (monotonic_ms(&now) != 0) {
            return -1;
        }

        if (now < current_sound_end_ms) {
            return 0;
        }

        current_sound_active = 0;
        current_sound_end_ms = 0;
    }

    if (!queue_pop(&next_sound)) {
```

```
        return 0;
    }

    return start_sound_now(next_sound);
}

void audio_clear_queue(void)
{
    queue_clear_only();
    current_sound_active = 0;
    current_sound_end_ms = 0;
}
```

(6) audio_interface.h

```
#ifndef AUDIO_INTERFACE_H
#define AUDIO_INTERFACE_H

#include "game.h"

int audio_init(void);
void audio_close(void);

int play_sound_id(sound_id_t sound);
int audio_enqueue_sound(sound_id_t sound);
int audio_submit_frame_sounds(sound_id_t sound1, sound_id_t sound2);
int audio_update(void);
void audio_clear_queue(void);
```

```
#endif
```

(7) vga_interface.c

```
#include "vga_interface.h"
#include "../kernel_modules/vga/vga_top.h"

#include <errno.h>
#include <fcntl.h>
#include <stdint.h>
#include <sys/ioctl.h>
#include <unistd.h>

#define PLAYER_SPRITE_X_ADJUST 2
#define PLAYER_SPRITE_Y_ADJUST 0
#define BOMB_SPRITE_X_ADJUST 2
#define BOMB_SPRITE_Y_ADJUST 0
#define UI_SPRITE_X_ADJUST 2
#define EXPLOSION_SPRITE_X_ADJUST 2

static uint16_t adjusted_coord(uint16_t value, int delta)
{
    int result = (int)value + delta;

    if (result < 0) {
        return 0;
    }

    if (result > 639) {
```

```
        return 639;
    }

    return (uint16_t)result;
}


#define VGA_DEVICE_PATH "/dev/vga_top"

#define PLAYER1_SPRITE_INDEX 0
#define PLAYER2_SPRITE_INDEX 1
#define BOMB_SPRITE_BASE_INDEX 2
#define EXPLOSION_SPRITE_BASE_INDEX 4
#define SPARE_SPRITE_BASE_INDEX 14
#define SPARE_SPRITE_COUNT 6
#define UI_SPRITE_BASE_INDEX 20
#define UI_SPRITE_COUNT 10

#define UI_NORMAL_X0 (192 + UI_SPRITE_X_ADJUST)
#define UI_NORMAL_Y 32
#define UI_GAME_OVER_X0 (192 + UI_SPRITE_X_ADJUST)
#define UI_GAME_OVER_Y 32

static int vga_fd = -1;

static uint8_t corrected_tile_col(uint8_t logical_col)
{
```

```
    if (logical_col == 0) {
        return MAP_COLS - 1;
    }

    return (uint8_t)(logical_col - 1);
}

static sprite_id_t player_sprite_for_direction(uint8_t player_index, direction_t dir)
{
    if (player_index == 0) {
        if (dir == DIR_UP) {
            return SPRITE_PLAYER1_BACK;
        }
        if (dir == DIR_LEFT) {
            return SPRITE_PLAYER1_LEFT;
        }
        if (dir == DIR_RIGHT) {
            return SPRITE_PLAYER1_RIGHT;
        }
        return SPRITE_PLAYER1_FRONT;
    }

    if (dir == DIR_UP) {
        return SPRITE_PLAYER2_BACK;
    }
    if (dir == DIR_LEFT) {
        return SPRITE_PLAYER2_LEFT;
```

```
    }
    if (dir == DIR_RIGHT) {
        return SPRITE_PLAYER2_RIGHT;
    }
    return SPRITE_PLAYER2_FRONT;
}

static sprite_id_t bomb_sprite_for_owner(uint8_t owner)
{
    return owner == 0 ? SPRITE_PLAYER1_BOMB :
SPRITE_PLAYER2_BOMB;
}

static sprite_id_t explosion_sprite_for_cell(uint8_t owner, int cell_index)
{
    if (owner == 0) {
        if (cell_index == 0) {
            return SPRITE_PLAYER1_EXPLOSION_CENTER;
        }
        if (cell_index == 1) {
            return SPRITE_PLAYER1_EXPLOSION_UP;
        }
        if (cell_index == 2) {
            return SPRITE_PLAYER1_EXPLOSION_DOWN;
        }
        if (cell_index == 3) {
            return SPRITE_PLAYER1_EXPLOSION_LEFT;
```

```
        }
        return SPRITE_PLAYER1_EXPLOSION_RIGHT;
    }

    if (cell_index == 0) {
        return SPRITE_PLAYER2_EXPLOSION_CENTER;
    }
    if (cell_index == 1) {
        return SPRITE_PLAYER2_EXPLOSION_UP;
    }
    if (cell_index == 2) {
        return SPRITE_PLAYER2_EXPLOSION_DOWN;
    }
    if (cell_index == 3) {
        return SPRITE_PLAYER2_EXPLOSION_LEFT;
    }
    return SPRITE_PLAYER2_EXPLOSION_RIGHT;
}

static int write_disabled_sprite(uint8_t sprite_index)
{
    sprite_t sprite;

    sprite.sprite_index = sprite_index;
    sprite.x = 0;
    sprite.y = 0;
    sprite.sprite_id = SPRITE_PLAYER1_FRONT;
```

```
    sprite.enable = 0;

    return write_sprite(sprite);
}

static int write_enabled_sprite(uint8_t sprite_index, uint16_t x, uint16_t y,
sprite_id_t sprite_id)
{
    sprite_t sprite;

    sprite.sprite_index = sprite_index;
    sprite.x = x;
    sprite.y = y;
    sprite.sprite_id = sprite_id;
    sprite.enable = 1;

    return write_sprite(sprite);
}

static uint8_t explosion_cell_visible(const game_state_t *game, const explosion_t
*explosion, int cell_index)
{
    uint8_t row = explosion->cells[cell_index].row;
    uint8_t col = explosion->cells[cell_index].col;

    if (row >= MAP_ROWS || col >= MAP_COLS) {
        return 0;
```

```
    }

    if (game->tiles[row][col] == TILE_HARD_WALL) {
        return 0;
    }

    return 1;
}

static int render_player(const game_state_t *game, uint8_t player_index)
{
    const player_t *player = &game->players[player_index];
    uint8_t sprite_index = player_index == 0 ? PLAYER1_SPRITE_INDEX :
PLAYER2_SPRITE_INDEX;

    if (!player->alive) {
        return write_disabled_sprite(sprite_index);
    }

    return write_enabled_sprite(
        sprite_index,
        adjusted_coord(player->x, PLAYER_SPRITE_X_ADJUST),
        adjusted_coord(player->y, PLAYER_SPRITE_Y_ADJUST),
        player_sprite_for_direction(player_index, player->dir)
    );
}
```

```
static int render_bomb(const game_state_t *game, uint8_t bomb_index)
{
    const bomb_t *bomb = &game->bombs[bomb_index];
    uint8_t sprite_index = (uint8_t)(BOMB_SPRITE_BASE_INDEX +
bomb_index);

    if (!bomb->active) {
        return write_disabled_sprite(sprite_index);
    }

    return write_enabled_sprite(
        sprite_index,
        adjusted_coord((uint16_t)(bomb->col * TILE_SIZE),
BOMB_SPRITE_X_ADJUST),
        adjusted_coord((uint16_t)(bomb->row * TILE_SIZE),
BOMB_SPRITE_Y_ADJUST),
        bomb_sprite_for_owner(bomb->owner)
    );
}

static int render_explosion(const game_state_t *game, uint8_t explosion_index)
{
    const explosion_t *explosion = &game->explosions[explosion_index];
    uint8_t base_index = (uint8_t)(EXPLOSION_SPRITE_BASE_INDEX +
explosion_index * EXPLOSION_TILE_COUNT);
    int i;
```

```
    if (!explosion->active) {
        for (i = 0; i < EXPLOSION_TILE_COUNT; ++i) {
            if (write_disabled_sprite((uint8_t)(base_index + i)) != 0) {
                return -1;
            }
        }
        return 0;
    }

    for (i = 0; i < EXPLOSION_TILE_COUNT; ++i) {
        uint8_t sprite_index = (uint8_t)(base_index + i);

        if (!explosion_cell_visible(game, explosion, i)) {
            if (write_disabled_sprite(sprite_index) != 0) {
                return -1;
            }
            continue;
        }

        if (write_enabled_sprite(
                sprite_index,
                  adjusted_coord((uint16_t)(explosion->cells[i].col * TILE_SIZE),
EXPLOSION_SPRITE_X_ADJUST),
                (uint16_t)(explosion->cells[i].row * TILE_SIZE),
                explosion_sprite_for_cell(explosion->owner, i)
            ) != 0) {
            return -1;
```

```
        }
    }

    return 0;
}

static int disable_spare_sprites(void)
{
    int i;

    for (i = 0; i < SPARE_SPRITE_COUNT; ++i) {
        if (write_disabled_sprite((uint8_t)(SPARE_SPRITE_BASE_INDEX + i)) !=
0) {
            return -1;
        }
    }

    return 0;
}

static int disable_ui_sprites(void)
{
    int i;

    for (i = 0; i < UI_SPRITE_COUNT; ++i) {
        if (write_disabled_sprite((uint8_t)(UI_SPRITE_BASE_INDEX + i)) != 0) {
            return -1;
```

```
        }
    }

    return 0;
}

static int render_normal_ui(void)
{
    if (write_enabled_sprite(20, UI_NORMAL_X0, UI_NORMAL_Y,
SPRITE_UI_PLAYER1) != 0) {
        return -1;
    }
    if (write_enabled_sprite(21, UI_NORMAL_X0 + 64, UI_NORMAL_Y,
SPRITE_UI_CHARACTER_V) != 0) {
        return -1;
    }
    if (write_enabled_sprite(22, UI_NORMAL_X0 + 128, UI_NORMAL_Y,
SPRITE_UI_CHARACTER_S) != 0) {
        return -1;
    }
    if (write_enabled_sprite(23, UI_NORMAL_X0 + 192, UI_NORMAL_Y,
SPRITE_UI_PLAYER2) != 0) {
        return -1;
    }

    return 0;
}
```

```
static int render_player1_win_ui(void)
{
   if (write_enabled_sprite(20, UI_GAME_OVER_X0, UI_GAME_OVER_Y,
SPRITE_UI_PLAYER1) != 0) {
      return -1;
   }
   if (write_enabled_sprite(21, UI_GAME_OVER_X0 + 64,
UI_GAME_OVER_Y, SPRITE_UI_YELLOW_W) != 0) {
      return -1;
   }
   if (write_enabled_sprite(22, UI_GAME_OVER_X0 + 128,
UI_GAME_OVER_Y, SPRITE_UI_YELLOW_I) != 0) {
      return -1;
   }
   if (write_enabled_sprite(23, UI_GAME_OVER_X0 + 192,
UI_GAME_OVER_Y, SPRITE_UI_YELLOW_N) != 0) {
      return -1;
   }

   return 0;
}

static int render_player2_win_ui(void)
{
   if (write_enabled_sprite(20, UI_GAME_OVER_X0, UI_GAME_OVER_Y,
SPRITE_UI_PLAYER2) != 0) {
```

```
        return -1;
    }
    if (write_enabled_sprite(21, UI_GAME_OVER_X0 + 64,
UI_GAME_OVER_Y, SPRITE_UI_PINK_W) != 0) {
        return -1;
    }
    if (write_enabled_sprite(22, UI_GAME_OVER_X0 + 128,
UI_GAME_OVER_Y, SPRITE_UI_PINK_I) != 0) {
        return -1;
    }
    if (write_enabled_sprite(23, UI_GAME_OVER_X0 + 192,
UI_GAME_OVER_Y, SPRITE_UI_PINK_N) != 0) {
        return -1;
    }

    return 0;
}

static int render_tie_ui(void)
{
    if (write_enabled_sprite(20, UI_GAME_OVER_X0, UI_GAME_OVER_Y,
SPRITE_UI_PLAYER1) != 0) {
        return -1;
    }
    if (write_enabled_sprite(21, UI_GAME_OVER_X0 + 64,
UI_GAME_OVER_Y, SPRITE_UI_CHARACTER_V) != 0) {
        return -1;
```

```
    }
    if (write_enabled_sprite(22, UI_GAME_OVER_X0 + 128,
UI_GAME_OVER_Y, SPRITE_UI_CHARACTER_S) != 0) {
        return -1;
    }
    if (write_enabled_sprite(23, UI_GAME_OVER_X0 + 192,
UI_GAME_OVER_Y, SPRITE_UI_PLAYER2) != 0) {
        return -1;
    }

    return 0;
}

static int render_ui(const game_state_t *game)
{
    if (disable_ui_sprites() != 0) {
        return -1;
    }

    if (!game->game_over) {
        return render_normal_ui();
    }

    if (game->winner == 0) {
        return render_player1_win_ui();
    }
```

```
    if (game->winner == 1) {
        return render_player2_win_ui();
    }

    return render_tie_ui();
}

int vga_init(void)
{
    if (vga_fd >= 0) {
        return 0;
    }

    vga_fd = open(VGA_DEVICE_PATH, O_RDWR);
    if (vga_fd < 0) {
        return -1;
    }

    if (disable_all_sprites() != 0) {
        close(vga_fd);
        vga_fd = -1;
        return -1;
    }

    return 0;
}
```

```
void vga_close(void)
{
    if (vga_fd >= 0) {
        close(vga_fd);
        vga_fd = -1;
    }
}

int write_tile(tile_t tile)
{
    struct vga_tile_arg arg;

    if (vga_fd < 0) {
        errno = ENODEV;
        return -1;
    }

    arg.row = tile.row;
    arg.col = tile.col;
    arg.tile_id = (uint8_t)tile.tile_id;
    arg.reserved = 0;

    return ioctl(vga_fd, VGA_WRITE_TILE, &arg);
}

int write_sprite(sprite_t sprite)
{
```

```
    struct vga_sprite_arg arg;

    if (vga_fd < 0) {
        errno = ENODEV;
        return -1;
    }

    arg.sprite_index = sprite.sprite_index;
    arg.sprite_id = (uint8_t)sprite.sprite_id;
    arg.enable = sprite.enable ? 1 : 0;
    arg.reserved = 0;
    arg.x = sprite.x;
    arg.y = sprite.y;

    return ioctl(vga_fd, VGA_WRITE_SPRITE, &arg);
}

int disable_all_sprites(void)
{
    if (vga_fd < 0) {
        errno = ENODEV;
        return -1;
    }

    return ioctl(vga_fd, VGA_DISABLE_ALL_SPRITES);
}
```

```
int vga_draw_initial_map(const tile_id_t tiles[MAP_ROWS][MAP_COLS])
{
    int row;
    int col;

    if (vga_fd < 0) {
        errno = ENODEV;
        return -1;
    }

    if (tiles == 0) {
        errno = EINVAL;
        return -1;
    }

    for (row = 0; row < MAP_ROWS; ++row) {
        for (col = 0; col < MAP_COLS; ++col) {
            tile_t tile;

            tile.row = (uint8_t)row;
            tile.col = corrected_tile_col((uint8_t)col);
            tile.tile_id = tiles[row][col];

            if (write_tile(tile) != 0) {
                return -1;
            }
        }
```

```
    }

    return 0;
}

int vga_render(const game_state_t *game)
{
    int i;

    if (game == 0) {
        errno = EINVAL;
        return -1;
    }

    if (vga_draw_initial_map(game->tiles) != 0) {
        return -1;
    }

    for (i = 0; i < PLAYER_COUNT; ++i) {
        if (render_player(game, (uint8_t)i) != 0) {
            return -1;
        }
    }

    for (i = 0; i < ACTIVE_BOMB_COUNT; ++i) {
        if (render_bomb(game, (uint8_t)i) != 0) {
            return -1;
```

```
        }
    }

    for (i = 0; i < ACTIVE_EXPLOSION_COUNT; ++i) {
        if (render_explosion(game, (uint8_t)i) != 0) {
            return -1;
        }
    }

    if (disable_spare_sprites() != 0) {
        return -1;
    }

    if (render_ui(game) != 0) {
        return -1;
    }

    return 0;
}
```

(8) vga_interfach

```
#ifndef VGA_INTERFACE_H
#define VGA_INTERFACE_H

#include "game.h"

int vga_init(void);
void vga_close(void);
```

```
int write_tile(tile_t tile);
int write_sprite(sprite_t sprite);

int disable_all_sprites(void);
int vga_draw_initial_map(const tile_id_t tiles[MAP_ROWS][MAP_COLS]);

int vga_render(const game_state_t *game);

#endif
```

(9) demo.c

```
#define _POSIX_C_SOURCE 200809L

#include "game.h"
#include "usbcontroller.h"
#include "vga_interface.h"
#include "audio_interface.h"

#include <errno.h>
#include <signal.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <time.h>

static volatile sig_atomic_t running = 1;
```

```
static void handle_signal(int signum)
{
   (void)signum;
   running = 0;
}

static void clear_inputs(controller_state_t inputs[PLAYER_COUNT])
{
   int i;

   for (i = 0; i < PLAYER_COUNT; ++i) {
      memset(&inputs[i], 0, sizeof(inputs[i]));
   }
}

static int sleep_one_tick(void)
{
   struct timespec req;
   struct timespec rem;

   req.tv_sec = 0;
   req.tv_nsec = 1000000000L / GAME_TICK_HZ;

   while (nanosleep(&req, &rem) != 0) {
      if (errno != EINTR) {
         return -1;
      }
```

```
        if (!running) {
            return 0;
        }
        req = rem;
    }

    return 0;
}

static void cleanup(void)
{
    audio_close();
    vga_close();
    usb_close();
}

int main(void)
{
    game_state_t game;
    controller_state_t inputs[PLAYER_COUNT];

    signal(SIGINT, handle_signal);
    signal(SIGTERM, handle_signal);

    clear_inputs(inputs);
    game_init(&game);
```

```
if (usb_init() != 0) {
    fprintf(stderr, "usb_init failed\n");
    cleanup();
    return 1;
}

if (usb_open() != 0) {
    fprintf(stderr, "usb_open failed\n");
    cleanup();
    return 1;
}

if (vga_init() != 0) {
    perror("vga_init");
    cleanup();
    return 1;
}

if (audio_init() != 0) {
    perror("audio_init");
    cleanup();
    return 1;
}

if (vga_draw_initial_map(game.tiles) != 0) {
    perror("vga_draw_initial_map");
    cleanup();
```

```
    return 1;
}

if (disable_all_sprites() != 0) {
    perror("disable_all_sprites");
    cleanup();
    return 1;
}

if (vga_render(&game) != 0) {
    perror("vga_render");
    cleanup();
    return 1;
}

while (running) {
    if (usb_read(inputs) != 0) {
        clear_inputs(inputs);
    }

    if (inputs[0].restart || inputs[1].restart) {
        audio_clear_queue();
    }

    game_step(&game, inputs);

    if (vga_render(&game) != 0) {
```

```
            perror("vga_render");
            break;
        }

        if (audio_submit_frame_sounds(game.pending_sound_1,
game.pending_sound_2) != 0) {
            perror("audio_submit_frame_sounds");
        }

        if (audio_update() != 0) {
            perror("audio_update");
        }

        if (sleep_one_tick() != 0) {
            perror("nanosleep");
            break;
        }
    }

    cleanup();
    return 0;
}
```

- kernel_modules(vga)

(1) vga_top.c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
```

```
#include <linux/miscdevice.h>
#include <linux/uaccess.h>
#include <linux/io.h>
#include <linux/mutex.h>
#include <linux/errno.h>
#include "vga_top.h"

#define VGA_TILE_PHYS 0xFF200000
#define VGA_TILE_SPAN 0x800
#define VGA_SPRITE_PHYS 0xFF201000
#define VGA_SPRITE_SPAN 0x80

#define VGA_MAP_ROWS 15
#define VGA_MAP_COLS 20
#define VGA_TILE_ID_COUNT 6
#define VGA_SPRITE_COUNT 30
#define VGA_SPRITE_ID_COUNT 30
#define VGA_SCREEN_WIDTH 640
#define VGA_SCREEN_HEIGHT 480

static void __iomem *tile_base;
static void __iomem *sprite_base;
static DEFINE_MUTEX(vga_mutex);

static int vga_check_ready(void)
{
    if (!tile_base || !sprite_base) {
```

```
        return -ENODEV;
    }
    return 0;
}

static void vga_write_tile_hw(const struct vga_tile_arg *arg)
{
    unsigned int index = arg->row * VGA_MAP_COLS + arg->col;
    unsigned int offset = 4 * index;
    iowrite32(arg->tile_id & 0x7, (u8 __iomem *)tile_base + offset);
}

static void vga_write_sprite_hw(const struct vga_sprite_arg *arg)
{
    u32 value;
    unsigned int offset = 4 * arg->sprite_index;

    value = (arg->x & 0x3ff)
          | ((arg->y & 0x1ff) << 10)
          | ((arg->sprite_id & 0x1f) << 19)
          | ((arg->enable ? 1u : 0u) << 24);

    iowrite32(value, (u8 __iomem *)sprite_base + offset);
}

static void vga_disable_all_sprites_hw(void)
{
```

```
    int i;

    if (!sprite_base) {
        return;
    }

    for (i = 0; i < VGA_SPRITE_COUNT; ++i) {
        iowrite32(0, (u8 __iomem *)sprite_base + 4 * i);
    }
}

static int vga_validate_tile(const struct vga_tile_arg *arg)
{
    if (arg->row >= VGA_MAP_ROWS) {
        return -EINVAL;
    }
    if (arg->col >= VGA_MAP_COLS) {
        return -EINVAL;
    }
    if (arg->tile_id >= VGA_TILE_ID_COUNT) {
        return -EINVAL;
    }
    return 0;
}

static int vga_validate_sprite(const struct vga_sprite_arg *arg)
{
```

```
    if (arg->sprite_index >= VGA_SPRITE_COUNT) {
        return -EINVAL;
    }
    if (arg->x >= VGA_SCREEN_WIDTH) {
        return -EINVAL;
    }
    if (arg->y >= VGA_SCREEN_HEIGHT) {
        return -EINVAL;
    }
    if (arg->sprite_id >= VGA_SPRITE_ID_COUNT) {
        return -EINVAL;
    }
    if (arg->enable > 1) {
        return -EINVAL;
    }
    return 0;
}

static long vga_ioctl(struct file *file, unsigned int cmd, unsigned long user_arg)
{
    int ret = 0;

    mutex_lock(&vga_mutex);

    ret = vga_check_ready();
    if (ret) {
        mutex_unlock(&vga_mutex);
```

```
    return ret;
}

switch (cmd) {
case VGA_WRITE_TILE:
{
    struct vga_tile_arg arg;

    if (copy_from_user(&arg, (void __user *)user_arg, sizeof(arg))) {
        ret = -EFAULT;
        break;
    }

    ret = vga_validate_tile(&arg);
    if (ret) {
        break;
    }

    vga_write_tile_hw(&arg);
    break;
}

case VGA_WRITE_SPRITE:
{
    struct vga_sprite_arg arg;

    if (copy_from_user(&arg, (void __user *)user_arg, sizeof(arg))) {
```

```
            ret = -EFAULT;
            break;
        }

        ret = vga_validate_sprite(&arg);
        if (ret) {
            break;
        }

        vga_write_sprite_hw(&arg);
        break;
    }

    case VGA_DISABLE_ALL_SPRITES:
        vga_disable_all_sprites_hw();
        break;

    default:
        ret = -EINVAL;
        break;
    }

    mutex_unlock(&vga_mutex);
    return ret;
}

static const struct file_operations vga_fops = {
```

```
    .owner = THIS_MODULE,
    .unlocked_ioctl = vga_ioctl,
    .llseek = no_llseek,
};

static struct miscdevice vga_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = VGA_DEVICE_NAME,
    .fops = &vga_fops,
    .mode = 0666,
};

static int __init vga_top_init(void)
{
    int ret;

    tile_base = ioremap(VGA_TILE_PHYS, VGA_TILE_SPAN);
    if (!tile_base) {
        return -ENOMEM;
    }

    sprite_base = ioremap(VGA_SPRITE_PHYS, VGA_SPRITE_SPAN);
    if (!sprite_base) {
        iounmap(tile_base);
        tile_base = NULL;
        return -ENOMEM;
    }
```

```
    ret = misc_register(&vga_misc_device);
    if (ret) {
        iounmap(sprite_base);
        iounmap(tile_base);
        sprite_base = NULL;
        tile_base = NULL;
        return ret;
    }

    vga_disable_all_sprites_hw();

    pr_info("vga_top loaded\n");
    return 0;
}

static void __exit vga_top_exit(void)
{
    vga_disable_all_sprites_hw();
    misc_deregister(&vga_misc_device);

    if (sprite_base) {
        iounmap(sprite_base);
        sprite_base = NULL;
    }

    if (tile_base) {
```

```
        iounmap(tile_base);
        tile_base = NULL;
    }

    pr_info("vga_top unloaded\n");
}

module_init(vga_top_init);
module_exit(vga_top_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("sy3342");
MODULE_DESCRIPTION("GridBrawl VGA kernel module");
```

(2) vga_top.h

```
#ifndef VGA_TOP_H
#define VGA_TOP_H

#ifdef __KERNEL__
#include <linux/types.h>
#include <linux/ioctl.h>
typedef u8 vga_u8;
typedef u16 vga_u16;
#else
#include <stdint.h>
#include <sys/ioctl.h>
typedef uint8_t vga_u8;
typedef uint16_t vga_u16;
```

```
#endif

#define VGA_DEVICE_NAME "vga_top"
#define VGA_IOCTL_MAGIC 'v'

struct vga_tile_arg {
    vga_u8 row;
    vga_u8 col;
    vga_u8 tile_id;
    vga_u8 reserved;
};

struct vga_sprite_arg {
    vga_u8 sprite_index;
    vga_u8 sprite_id;
    vga_u8 enable;
    vga_u8 reserved;
    vga_u16 x;
    vga_u16 y;
};

#define VGA_WRITE_TILE _IOW(VGA_IOCTL_MAGIC, 1, struct
vga_tile_arg)
#define VGA_WRITE_SPRITE _IOW(VGA_IOCTL_MAGIC, 2, struct
vga_sprite_arg)
#define VGA_DISABLE_ALL_SPRITES _IO(VGA_IOCTL_MAGIC, 3)
```

#endif

- kernel_modules(audio)

(1) fpga_audio.c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/miscdevice.h>
#include <linux/uaccess.h>
#include <linux/io.h>
#include <linux/mutex.h>
#include <linux/errno.h>
#include "fpga_audio.h"

#define FPGA_AUDIO_PHYS 0xFF202000
#define FPGA_AUDIO_SPAN 0x4

static void __iomem *audio_base;
static DEFINE_MUTEX(fpga_audio_mutex);

static ssize_t fpga_audio_write(struct file *file, const char __user *buf, size_t
count, loff_t *ppos)
{
    fpga_audio_u32 sound_id;
    ssize_t ret;

    if (count != sizeof(sound_id)) {
        return -EINVAL;
```

```
    }

    if (copy_from_user(&sound_id, buf, sizeof(sound_id))) {
        return -EFAULT;
    }

    mutex_lock(&fpga_audio_mutex);

    if (!audio_base) {
        ret = -ENODEV;
        goto out;
    }

    if (sound_id == FPGA_AUDIO_SOUND_NONE) {
        ret = sizeof(sound_id);
        goto out;
    }

    if (sound_id < FPGA_AUDIO_MIN_SOUND_ID || sound_id >
FPGA_AUDIO_MAX_SOUND_ID) {
        ret = -EINVAL;
        goto out;
    }

    iowrite32(sound_id, audio_base);
    ret = sizeof(sound_id);
```

```
out:
    mutex_unlock(&fpga_audio_mutex);
    return ret;
}

static const struct file_operations fpga_audio_fops = {
    .owner = THIS_MODULE,
    .write = fpga_audio_write,
    .llseek = no_llseek,
};

static struct miscdevice fpga_audio_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = FPGA_AUDIO_DEVICE_NAME,
    .fops = &fpga_audio_fops,
    .mode = 0666,
};

static int __init fpga_audio_init(void)
{
    int ret;

    audio_base = ioremap(FPGA_AUDIO_PHYS, FPGA_AUDIO_SPAN);
    if (!audio_base) {
        return -ENOMEM;
    }
```

```
    ret = misc_register(&fpga_audio_misc_device);
    if (ret) {
        iounmap(audio_base);
        audio_base = NULL;
        return ret;
    }

    pr_info("fpga_audio loaded\n");
    return 0;
}

static void __exit fpga_audio_exit(void)
{
    misc_deregister(&fpga_audio_misc_device);

    if (audio_base) {
        iounmap(audio_base);
        audio_base = NULL;
    }

    pr_info("fpga_audio unloaded\n");
}

module_init(fpga_audio_init);
module_exit(fpga_audio_exit);

MODULE_LICENSE("GPL");
```

```
MODULE_AUTHOR("sy3342");
MODULE_DESCRIPTION("GridBrawl FPGA audio kernel module");
```

(2) fpga_audio.h

```
#ifndef FPGA_AUDIO_H
#define FPGA_AUDIO_H

#ifdef __KERNEL__
#include <linux/types.h>
typedef u32 fpga_audio_u32;
#else
#include <stdint.h>
typedef uint32_t fpga_audio_u32;
#endif

#define FPGA_AUDIO_DEVICE_NAME "fpga_audio"

#define FPGA_AUDIO_SOUND_NONE 0u
#define FPGA_AUDIO_MIN_SOUND_ID 1u
#define FPGA_AUDIO_MAX_SOUND_ID 4u

#endif
```