FPGA-Driven 'Forest Fire and Ice' Game: Design and Implementation

Yifan Mao
ym3064@columbia.edu

Weijie Wang
ww2739@columbia.edu

Yonghao Lin
yl5763@columbia.edu

Yang Cao
yc4535@columbia.edu

Zhenqi Li
zl3508@columbia.edu

Contents

1 Introduction

This project aims to implement a simplified version of the popular cooperative puzzle-platformer game *Forest Fire and Ice* on the Terasic DE1-SoC FPGA platform. The game features two characters—Fire Boy and Water Girl—who must navigate through a series of levels filled with obstacles, traps, and interactive elements.

The system leverages both the FPGA fabric and the Hard Processor System (HPS) of the Cyclone V SoC. The FPGA is responsible for generating 640×480 @ 60Hz VGA video output, performing real-time tile-and-sprite composition and optional collision detection, and streaming PCM audio. Meanwhile, the dual-core ARM processor executes high-level game logic, player control, and system coordination.

Gameplay is controlled via a USB input device (i.e. keyboard), with the video output rendered to a VGA monitor. The game is designed to run at a steady 60 frames per second to ensure smooth and responsive interaction.

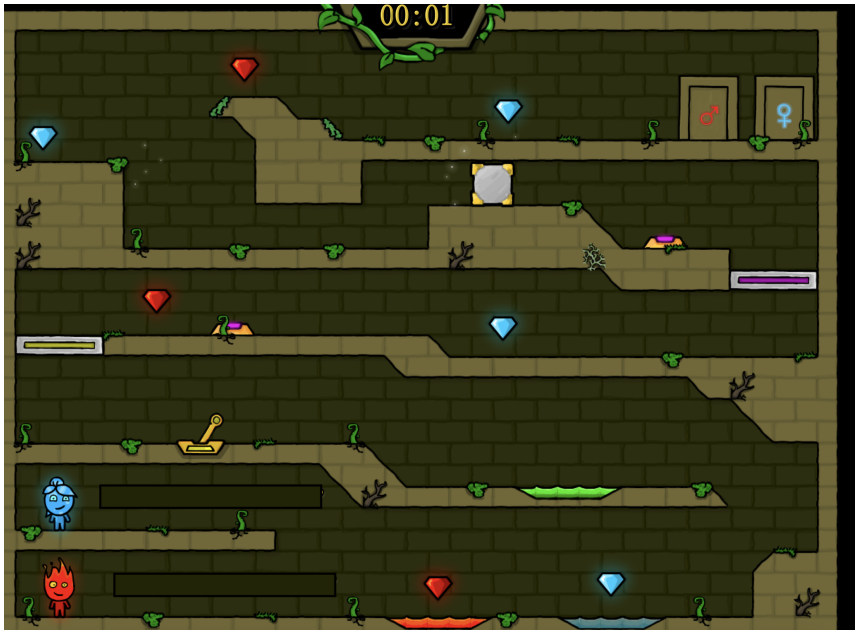

Figure 1: Game of "Forest Fire and Ice"

2 System Block Diagram

This system clearly partitioned into two domains:

- **Hardware Domain (FPGA / SystemVerilog)** — handles all pixel-rate activities (tile+sprite mixing, optional collision detector, audio serialization, VGA timing).
- **Software Domain (ARM Core / C)**: Focuses on high-level game logic such as collision detection, rule enforcement, and player state transitions.

Communication between the software and hardware domains is conducted via the **Avalon Bus**, a memory-mapped interface that allows the ARM processor to write/read registers/BRAM in the FPGA.

2.1 Data Communication Design

To maximize performance and maintain modularity, all rendering is handled by the FPGA hardware. The software domain is responsible only for transmitting high-level game state to the hardware. Each sprite (e.g., Fire Boy, Water Girl, game objects) is described by a fixed-size entry in the memory-mapped `SPRITE_ATTR_TABLE`, updated once per frame.

Each sprite entry includes the following fields:

- **X, Y Coordinates**: The on-screen position of the sprite in pixel space.

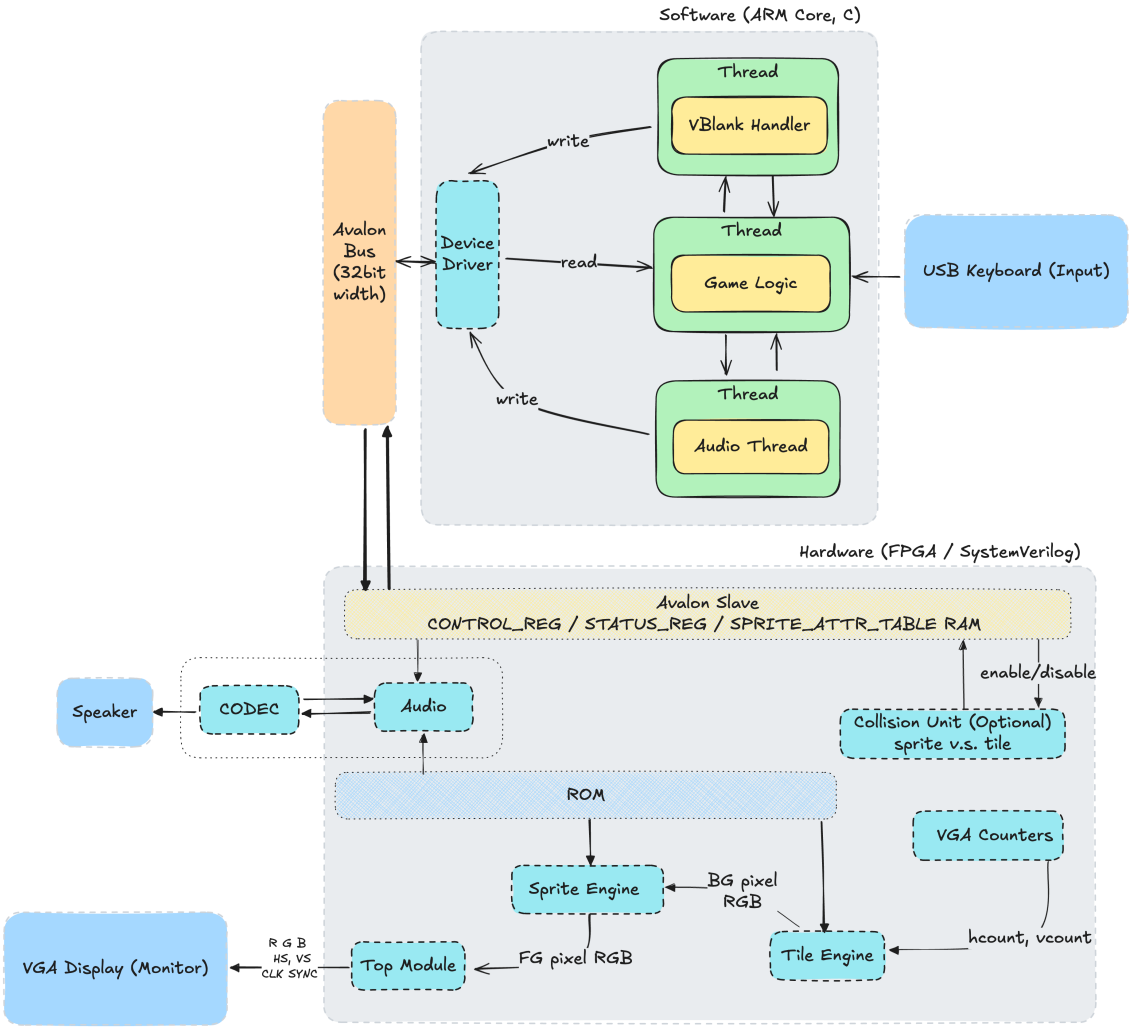

Figure 2: Block Diagram of "Forest Fire and Ice"

- **Frame ID**: An index into the sprite pattern ROM, used to select the animation frame.
- **Priority**: Determines z-ordering; higher priority sprites (e.g., Water Girl) visually cover lower priority ones (e.g., Fire Boy).
- **Width, Height**: Defines the bounding box of the sprite for rendering and hardware collision.
- **Flags**: Encoded status such as alive/dead, jumping, or on a switch.

These sprite attributes are written by the software to the MMIO `SPRITE_ATTR_TABLE`, which is stored in FPGA on-chip RAM and used directly by the rendering and collision engines.

2.2 Hardware Responsibilities

- Generates synchronized VGA signals (HSYNC, VSYNC, RGB) for real-time video output.
- Stores tile maps, sprite patterns, and audio samples in on-chip BRAM.
- Provides MMIO-based register interfaces (CONTROL_REG, STATUS_REG) accessible from HPS.
- Selects and renders background tiles using a Tile Engine based on CONTROL_REG selection.
- Manages sprite rendering and composition over tiles via the Sprite Engine.
- Performs optional hardware-accelerated collision detection between sprites and tilemap elements.
- Outputs PCM audio data stored in BRAM via I2S interface to WM8731 codec.

2.3 Software Responsibilities

The software periodically polls input devices and executes all high-level game logic. Its responsibilities include:

- Parsing user input (e.g., from USB keyboard) to update player movement and game state.
- Handling interactions with environment elements such as doors, lever, and switches.
- Detecting sprite-to-sprite collisions in software.
- Optionally handling sprite-to-tile collisions in software, or enabling the hardware collision detector via `CONTROL_REG[2]`.
- Polling the `STATUS_REG` to read VGA scanline (`vcount`) and safely write updates during the VBlank window (i.e., when `vcount` ≥ 480).
- Updating the `CONTROL_REG` to select the current tilemap and trigger audio playback.

By separating game logic from display logic, and synchronizing all MMIO writes during the vertical blanking interval, the software ensures smooth, tear-free frame rendering at 60Hz.

3 Algorithms

3.1 Software

3.1.1 Character Movement Logic

Each character—Fireboy and Watergirl—has distinct movement capabilities, managed through a finite state machine (FSM) that interprets player inputs and updates character states accordingly.
Movement States:

1. Idle: No movement input detected.
2. Moving Left/Right: Horizontal movement based on input.
3. Jumping: Initiated when the jump input is received, and the character is grounded.
4. Falling: Occurs when the character is airborne without upward velocity.

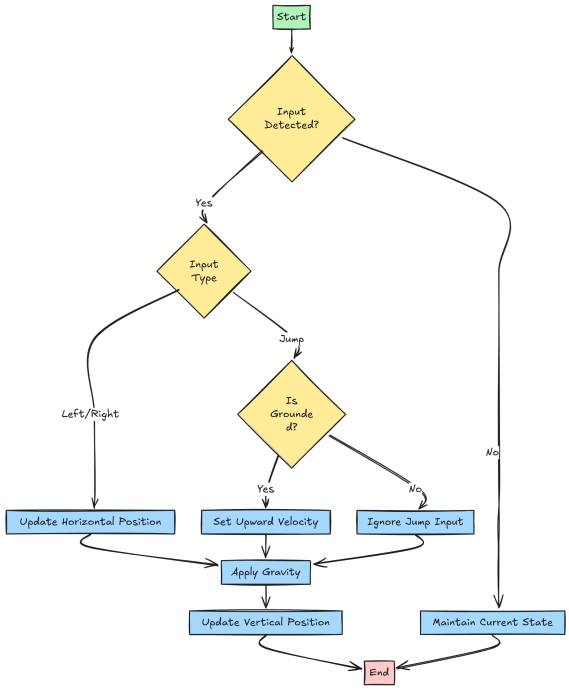

Figure 3: The flowchart of Character Movement Logic

Gravity is simulated by incrementally increasing downward velocity when the character is airborne. And Post-movement, collision detection ensures characters do not pass through solid objects.

3.1.2 Collision Detection Algorithm

Collision detection ensures that characters interact appropriately with the game environment, including platforms, hazards, and interactive objects. We utilize Axis-Aligned Bounding Box (AABB) collision detection, comparing the rectangular boundaries of characters and objects. Collision Response Examples:

1. Solid Objects: Prevent movement through the object by resetting position to the last valid state.
2. Hazards (e.g., Water for Fireboy, Lava for Watergirl): Trigger character death sequence.
3. Interactive Objects (e.g., Buttons, Levers): Activate associated mechanisms.

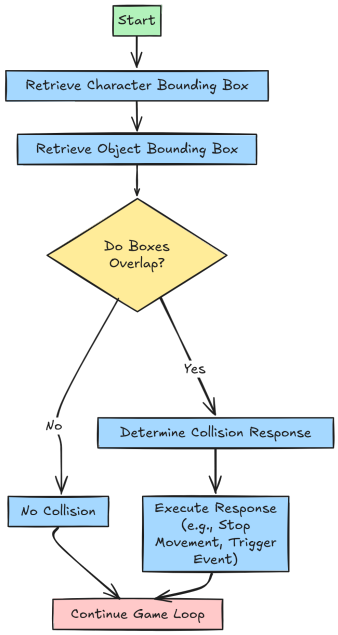

Figure 4: The flowchart of Collision Detection

3.1.3 Interactive Mechanisms Logic

Interactive mechanisms, such as buttons and levers, alter the game environment when activated by characters. For example, When pressed the button, may open doors or move platforms or toggle states of environmental elements, such as activating traps or elevators.

Mechanism Activation Conditions:

- Proximity: Character's bounding box overlaps with the mechanism's bounding box.
- Action Input: Specific input received to activate the mechanism (if required)

3.2 Hardware

The hardware components implement the following algorithms to efficiently handle graphics rendering, sprite management, collision detection, and audio playback:

3.2.1 VGA Video Generation Algorithm

- The VGA controller maintains horizontal (`hcount`) and vertical (`vcount`) counters synchronized to generate a standard 640×480 @60Hz VGA output.
- On each clock cycle, these counters provide coordinates (x, y) to the Tile and Sprite Engines to determine the pixel colors for display.

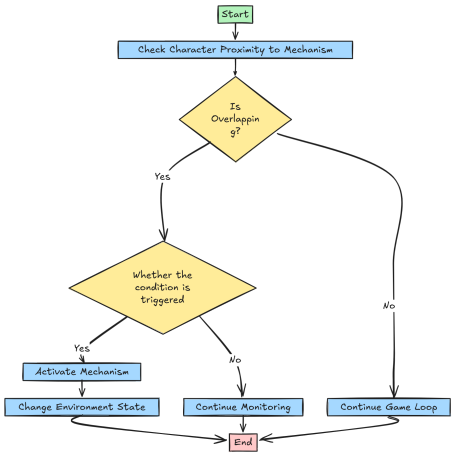

Figure 5: The flowchart of Interactive Mechanism

3.2.2 Tile Rendering Algorithm

- The active tile map (selected by the software via `CONTROL_REG`) is stored in FPGA on-chip ROM (three preloaded tile maps, each 40×30 tiles).
- Given the current (x, y) scan coordinates, the Tile Engine calculates which tile should appear at this pixel location:

$$\begin{aligned} tile\_x &= x/16 && (x >> 4) \\ tile\_y &= y/16 && (y >> 4) \\ tile\_id &= TILE\_MAP\_ROM[active\_map][tile\_y][tile\_x] \end{aligned}$$

- The pixel within the tile is determined by:

$$\begin{aligned} pixel\_x &= x\%16 && (x\&0xF) \\ pixel\_y &= y\%16 && (y\&0xF) \\ color\_id &= TILE\_PATTERN\_ROM[tile\_id][pixel\_y][pixel\_x] \end{aligned}$$

- `color_id` is used to lookup RGB values from the unified Color Palette ROM (RGB888).

3.2.3 Sprite Rendering and Composition Algorithm

- Up to 16 sprites are stored in an FPGA BRAM buffer (`SPRITE_ATTR_TABLE`), updated each frame by software via MMIO.
- Each sprite entry contains $\{x, y, frame\_id, priority, width, height, flags\}$ (8 Bytes).
- For each pixel (x, y):
  - The Sprite Engine iterates through all sprites to determine coverage at (x, y):
    - Checks if (x, y) lies within a sprite's bounding box:

$$\begin{aligned} sprite\_x \le x < sprite\_x + width \\ sprite\_y \le y < sprite\_y + height \end{aligned}$$

    - If multiple sprites overlap at this pixel, select the sprite pixel color ID from the highest priority sprite (larger `priority` value indicates higher priority).
    - In the default scenario, the Water Girl sprite has higher priority than Fire Boy, meaning Water Girl can visually cover Fire Boy when overlapped, and no collision between them is detected due to priority handling.
    - Fetch the sprite pixel color ID:

$$\begin{aligned} pixel\_x &= x - sprite\_x \\ pixel\_y &= y - sprite\_y \\ color\_id &= SPRITE\_PATTERN\_ROM[frame\_id][pixel\_y][pixel\_x] \end{aligned}$$

    * If `color_id` is non-zero (non-transparent), this pixel overrides any lower priority sprite or tile pixel at this location.

- The final composited pixel RGB value is sent directly to the VGA output.

3.2.4 Collision Detection Algorithm

- Hardware-based collision detection is an optional feature that can be enabled or disabled by software via the collision enable bit in `CONTROL_REG`. Software can choose to handle collision detection entirely in software if desired.
- When enabled, the hardware Collision Detector checks for collisions between player sprites and tilemap elements only. Collision detection between sprites must be implemented fully in software. Additionally, software can optionally implement sprite-to-tile collision detection without using the hardware unit.
- For player sprite bounding boxes, the hardware Collision Detector performs the following steps:
  - Computes player bounding boxes based on sprite attributes (`x, y, width, height`).
  - Samples relevant tile IDs from the active tile map ROM at positions overlapped by the sprite bounding boxes.
  - Determines collision type by comparing sampled tile IDs against predefined tile types (`FIRE`, `WATER`, `WALL`, `POISON`, `GOAL`).
- Specifically:
  - Water Girl sprite will be detected as collided (and thus will "die") if overlapping a `FIRE` tile.
  - Fire Boy sprite will similarly "die" if overlapping a `WATER` tile.
  - Both Water Girl and Fire Boy sprites will "die" if overlapping a `POISON` tile.
- Collision detection results (`collision type`) are written to `STATUS_REG` bits [23:20] for software to read and respond accordingly.

3.2.5 Audio Playback Algorithm

- Audio clips are preloaded into BRAM as 8kHz, 16-bit PCM mono samples.
- Software selects audio clip playback by writing the audio select bits in `CONTROL_REG`.
- FPGA continuously streams selected audio clip data through an internal FIFO to the WM8731 audio codec via I2S.
- Audio FIFO ensures smooth, uninterrupted playback without CPU intervention.

These algorithms are fully pipelined and synchronized using a unified system clock, ensuring smooth and continuous rendering, accurate collision detection, and seamless audio playback.

4 Resource Budgets

The DE1-SoC platform provides limited on-chip FPGA resources, particularly in terms of BRAM (Block RAM), logic elements (LEs), and DSP blocks. This section summarizes the estimated consumption of these limited resources, based on the current system architecture.

4.1 On-Chip Memory (BRAM) Usage

Component	Memory Type	Size Estimate	Description
Tile Maps (x3)	BRAM (ROM)	3.6 KB	3 static tilemaps, each 40×30 tiles (1 byte per tile)
Tile Pattern Table	BRAM (ROM)	64 KB	256 tile types, each 16×16 pixels, 8-bit color ID
Sprite Pattern Table	BRAM (ROM)	32 KB	Multiple sprite animation frames, 8-bit color ID per pixel
Sprite Attribute Table	BRAM (RAM)	128 B	16 sprite entries, each 8 bytes
Color Palette	BRAM (ROM)	768 B	256 entries, each 24-bit RGB888
Audio Clip ROM	BRAM (ROM)	160 KB	≤10 seconds of mono 8kHz 16-bit PCM audio
Audio FIFO	BRAM (FIFO)	1 KB	256 entries × 32-bit word depth for smooth streaming
Collision Status Buffer	Register	negligible	Result flags stored in STATUS_REG
MMIO Control Registers	Register	negligible	CONTROL_REG, STATUS_REG, etc.
Total		**∼261 KB**	Well within ∼495 KB BRAM budget of the Cyclone V FPGA

Table 1: Estimated On-Chip BRAM Usage

Table 2 below summarizes the estimated ROM usage for all sprites in the game, including characters, items, and interactive objects. Each frame pixel is stored using 8-bit color-ID to retrieve RGB888 from Color Palette.

Sprite	Frame Size	Bytes/Frame	Frames	Total Memory (Bytes)	Description
Fireboy	2x16x16	2x256	2x10	10240	stand, walk, run (left/right)
Watergirl	2x16x16	2x256	2x10	10240	stand, walk, run (left/right)
Red Gem	16x16	256	1	256	Collection, disappears after collision
Blue Gem	16x16	256	1	256	Collection, disappears after collision
Purple Button	16x16	256	2	512	Pressed/unpressed animation states
Yellow Elevator	32x16	512	2	1024	Moving platform (up/down) animation
Purple Elevator	32x16	512	2	1024	Moving platform (up/down) animation
Yellow Lever	16x16	256	2	512	Toggleable lever: left/right state
Fire Pool	32x16	512	2	1024	Hazardous to Watergirl only
Water Pool	32x16	512	2	1024	Hazardous to Fireboy only
Green Pool	32x16	512	2	1024	Hazardous to all characters
Box	32x32	1024	1	1024	Movable block used in puzzles

Table 2: Detailed Sprite Pattern Table with Memory Breakdown

The total ROM usage for all sprite patterns is approximately **28 KB**, which is well within the 32 KB allocated in BRAM for the Sprite Pattern Table. This leaves ample room for adding new animation frames or additional sprite types in future iterations.

4.2 Logic Resources (LEs and Control)

- **Tile Engine:** ∼4,000 LEs — tilemap lookup, tile pattern access, palette decoding.
- **Sprite Engine:** ∼5,000 LEs — per-pixel sprite overlay logic, priority comparison.
- **VGA Controller:** ∼2,000 LEs — synchronization counters and pixel timing generator.
- **Collision Unit:** ∼2,000 LEs — tile sampling, bounding box comparator.
- **Audio Output:** ∼2,000 LEs — I2S interface, FIFO control, I2C codec setup.
- **Register Interface/MMIO:** ∼1,000 LEs — Avalon-MM slave logic, register decoding.

Estimated total: ∼16,000 LEs — comfortably within the ∼50K logic elements available on the DE1-SoC FPGA.

4.3 DSP and Multiplier Resources

- The design avoids use of hardware multipliers or DSP blocks.
- All tile and sprite positioning is handled via simple integer arithmetic (division by powers of two and modulo).
- Audio playback is stream-based, with no filtering or mixing done in hardware.

4.4 External Memory and Bandwidth Considerations

- All visual and audio assets are fully preloaded into BRAM at boot time — there is **no runtime streaming from HPS DDR3**.
- Tilemaps and tile/sprite patterns are static during runtime; only sprite attribute table updates (128B per frame) are written via MMIO.
- Audio FIFO consumes low bandwidth and is software-controlled; clip selection does not require data fetching from SD or DDR3 after boot.

4.5 Summary

The design fits comfortably within the on-chip BRAM and logic resource limits of the Cyclone V FPGA on the DE1-SoC. By avoiding streaming or dynamic asset loading and using pipelined engines for rendering, the system guarantees low-latency output while staying resource-efficient. The hardware collision unit and audio output system are optional accelerators, offloading lightweight tasks while leaving core logic in software control.

5 The Hardware/Software Interface

This section details the MMIO-based interface between the HPS software and the custom FPGA hardware. All communication between the two sides occurs through memory-mapped registers and shared RAM regions, exposed via an Avalon-MM slave interface mapped into the HPS address space. The interface allows software to control rendering, sprite positioning, collision logic, and audio playback.

5.1 Memory-Mapped Register Summary

Register Name	Address Offset	Width	R/W	Description
`CONTROL_REG`	0x000	32-bit	Write-only	**[1:0]**: Tilemap select (0–2) **[2]**: Enable hardware collision detection (1 = enable) **[4:3]**: Audio clip select (0–3) Unused bits should be written as zero.
`STATUS_REG`	0x004	32-bit	Read-only	**[9:0]**: Current horizontal VGA pixel count **[19:10]**: Current vertical VGA pixel count **[23:20]**: Collision result type - 0: No collision or detection disabled - 1: Any sprite touched WALL tile - 1: Water Girl touched FIRE tile - 2: Fire Boy touched WATER tile - 3: Any sprite touched POISON tile - 4: Player reached GOAL tile
`SPRITE_ATTR_TABLE`	0x700–0x77F	128 bytes	R/W	16 entries × 8 bytes per sprite. Each sprite entry contains: **[15:0]**: `x` position (in pixels) **[31:16]**: `y` position (in pixels) **[39:32]**: `frame_id` (animation frame index) **[43:40]**: `priority` (used for pixel overlay condition) **[47:44]**: `reserved flags` (bitfield, reserved for future features) **[55:48]**: `width` (bounding box width in pixels) **[63:56]**: `height` (bounding box height in pixels)

Table 3: Hardware-Controlled MMIO Registers

5.2 Communication Protocol

- The HPS software updates the `SPRITE_ATTR_TABLE` at each VBlank period, typically once per frame.
- The software selects which tilemap and audio clip to use by writing to `CONTROL_REG`.
- Hardware updates the `STATUS_REG` every clock cycle with real-time scan position and collision results.
- Collision detection is performed in hardware only when enabled, and only for sprite-to-tilemap interactions. Sprite-to-sprite collisions are fully handled in software.
- Audio playback is initiated in hardware when a new clip index is written to `CONTROL_REG[4:3]`. The corresponding clip is streamed from BRAM to the I2S interface.

5.3 Access Mechanism

- All registers and MMIO buffers are exposed via an Avalon-MM slave peripheral. A corresponding Linux device driver will be implemented to provide structured access from user-space programs via standard interfaces such as `ioctl()` or `mmap()`, with the driver using `of_iomap()` internally to map device registers. Alternatively, `/dev/mem` may be used for low-level access during development.

- Software should align all 32-bit writes and reads to word boundaries and avoid partial byte writes.
- Updates to the sprite attribute table must be written atomically to avoid visual glitches.

5.4 Polling-Based Synchronization

- This system does not currently use interrupts for synchronization. Instead, the HPS software polls the `STATUS_REG` each frame to track VGA timing.
- Specifically, the software reads the `vcount` value from `STATUS_REG[19:10]`.
- While $0 \leq$ `vcount` $<$ `480`, rendering is in progress; software prepares updated data (sprites, control register).
- When `vcount` $\geq$ `480` (during VBlank), software performs writes to hardware (e.g., sprite attribute table, CONTROL_REG). This avoids visual tearing by ensuring all updates take effect at the start of the next frame.
- This polling-based synchronization is simple and effective for the game's frame-locked rendering loop.