The Design Document for CSEE 4840 Embedded System Design
Street Fighting Game

Team members:
Yang Li yl6070, Jiyang Yin jy3557, Zhewen Guo zg2567

Spring 2026

1 Introduction

1.1 Problem Statement and Project Goal

This project aims to build a two-player fighting game on the DE1-SoC FPGA platform. The system allows two players to control fighters using USB keyboards, while the game logic, combat resolution, rendering, and audio are split between software running on the HPS and hardware implemented on the FPGA.

The current goal is to create a playable prototype that supports a full fighting-game loop, including menu state, gameplay state, attacks, blocking, damage, KO, timeout, and audio feedback. The longer-term goal is to move video rendering from the current HPS software path into an FPGA VGA rendering module.

1.2 Motivation and Importance

This project is meaningful for several reasons:

- **Embedded System Design:** It combines HPS software, FPGA hardware, memory-mapped I/O, and real-time interaction in one system.
- **Game Logic and Hardware Co-Design:** The project demonstrates how high-level game logic can remain in software while timing-sensitive output paths are moved into hardware.
- **Real-Time User Interaction:** Two-player keyboard input, real-time collision, and frame-based updates make this a responsive interactive system rather than a static display demo.
- **Incremental FPGA Development:** The project already contains a working gameplay path and an implemented FPGA audio peripheral, while leaving room for future FPGA VGA acceleration.

1.3 Scope and Key Modules

This project separates responsibilities between software and hardware.

Software (HPS side):

- USB keyboard input capture and parsing
- Game-state management
- Fighter state updates
- Collision detection and combat logic
- Timer, health, KO, and match flow
- Current software rendering to `/dev/fb0` or console

Hardware (FPGA side):

- FPGA-backed audio peripheral for the WM8731 codec
- Avalon-MM interface for audio data and status
- Future extension path for VGA rendering through a dedicated MMIO-driven peripheral

2 System Block Diagram

2.1 Overall System Block Diagram

The current top-level data path is shown below:

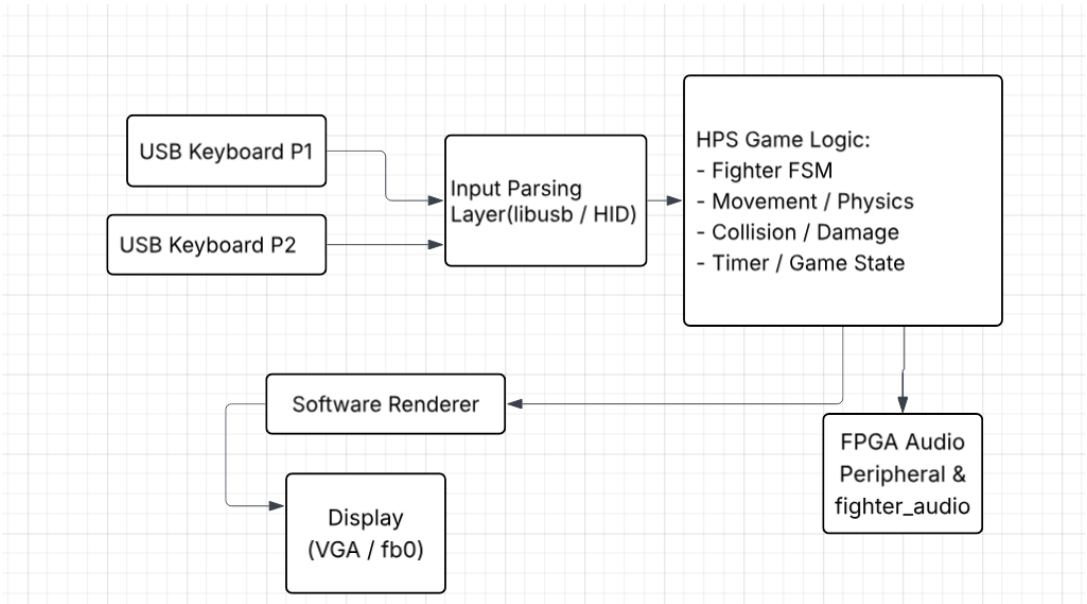

Figure 1: Top-level system architecture

2.2 Hardware Components Explanation

1. **Quartus Project and Verilog Source Files:** The FPGA side includes the board-level top module and the custom audio peripheral. The current Quartus build connects the audio hardware to the board pins, while VGA outputs remain disabled in the current implementation.

2. **FPGA Audio Peripheral:** The custom module `fighter_audio_wm8731` implements Avalon-MM registers, FIFOs, codec initialization through I2C, and audio sample serialization for the WM8731 codec.

3. **WM8731 Codec Interface:** The FPGA drives the codec clocks and serial audio path. This makes audio the first completed custom hardware output path in the project.

2.3 Software Components Explanation

1. **Game Logic in C:** The HPS runs the main fighting-game logic, including menu state, gameplay state, round timing, attack phases, blocking, damage, KO handling, and game-over conditions.

2. **Renderer in C:** The current renderer draws the menu, gameplay background, player graphics, health bars, timer, and debug overlays through a software framebuffer path. If the framebuffer is unavailable, the program falls back to console output.

3. **USB Keyboard Driver Layer:** USB HID keyboard devices are detected and polled through `libusb`. Up to two keyboards are supported.

4. **Audio Command Layer:** Software generates audio events such as menu music and game-over sounds, then sends them either to the FPGA-backed WM8731 path or to a command-based fallback backend.

3 Algorithms

3.1 Game Loop Algorithm (in software)

Each frame, the software executes the following steps:

1. Read or synthesize player input
2. Parse input into normalized player commands
3. Update the global game state
4. Resolve movement, attacks, blocking, and collisions
5. Update HP, timer, and match result
6. Generate audio events if needed
7. Draw the current frame

8. Sleep briefly to maintain approximately 60 FPS

This loop allows the project to keep all gameplay logic in one place while still supporting hardware-backed outputs.

3.2 Fighter State Machine Algorithm (in software)

Each player is represented by a state machine. The major states include:

- `IDLE`
- `WALK`
- `JUMP`
- `CROUCH`
- `GUARD`
- `ATTACK`
- `HIT`
- `BLOCK_STUN`
- `KO`

The state machine changes according to player input, frame timing, and combat results. For example:

- pressing movement keys updates walking or jumping states
- pressing attack triggers an attack command and enters attack phases
- successful collision causes the opponent to enter hit or block-stun states
- health reaching zero causes a KO transition

3.3 Combat Resolution Algorithm (in software)

Combat is handled by checking attack state, player position, facing direction, and overlap rules.

1. Determine whether a player is in an active attack phase
2. Compute whether the attack range overlaps the opponent
3. If overlap occurs, classify the result as hit, blocked, trade, or whiff
4. Apply HP reduction, hit stun, or block stun
5. Record event flags and combat result for rendering/debug purposes

The project supports several attack commands, including normal attack, fireball, dragon punch, jump attack, forward jump attack, back jump attack, and sweep.

3.4 Rendering Algorithm (current software path)

The current renderer does not yet use a custom VGA hardware module. Instead, it uses a software rendering approach:

1. Determine the current global game state
2. If in menu state, load and display menu-frame images
3. If in gameplay state, draw background, health bars, timer, player graphics, and debug overlays
4. Output the frame through `/dev/fb0` or console

This software rendering stage is useful because it allows the team to validate gameplay behavior before implementing FPGA VGA rendering.

3.5 Audio Playback Algorithm (hardware + software)

Audio playback is split between software and hardware:

1. The HPS generates commands such as `PLAY_ONCE`, `START_LOOP`, and `STOP_LOOP`
2. Software writes sample data and control information through MMIO
3. The FPGA audio peripheral stores sample words in FIFOs
4. The peripheral initializes the WM8731 through I2C
5. The FPGA serializes audio data and sends it to the codec

3.6 Fighter Visual State Machine

The visual behavior of each fighter is controlled by a priority-based finite state machine (FSM). This FSM determines which animation or visual state should be displayed at each frame, based on the current game state, player input, and combat events.

Unlike a traditional transition-based FSM, this implementation evaluates conditions in a fixed priority order. The first satisfied condition determines the current visual state. This design simplifies state management and ensures that higher-priority states, such as KO or HIT, override lower-priority states.

Current State	Trigger Condition / Input	Next State	Description
Any	`hp <= 0`	KO	Character dies and enters KO state
Any	`hurt_visual_frames > 0`	HIT	Briefly displays HIT after being hit
Any	`attack_visual_frames > 0`	ATTACK	Displays ATTACK during attack animation
Any	`player->y < ground_y OR player->vy != 0`	JUMP	Character is airborne
Any	`input->guard_held`	GUARD	Player is holding guard button
Any	`input->crouch_held`	CROUCH	Player is holding crouch button
Any	`input->move_left OR input->move_right`	WALK	Player is moving horizontally
Any	Otherwise	IDLE	Default idle state

Table 1: Fighter Visual State Machine

3.6.1 FSM Design Notes

- **Priority-based evaluation:** The FSM does not rely on explicit transitions between states. Instead, conditions are checked in priority order, and the first satisfied condition determines the state.
- **KO has the highest priority:** Once `hp <= 0`, the fighter remains in the KO state regardless of input or other conditions.
- **HIT overrides most states:** When a fighter takes damage, the `HIT` state temporarily overrides movement and guard states.
- **ATTACK state timing:** The `ATTACK` state is controlled by `attack_visual_frames`, ensuring consistent animation during attack sequences.
- **Airborne detection:** The `JUMP` state is active whenever the player is not grounded (`vy (Vertical velocity) != 0` or above ground level).
- **Ground states:** `GUARD`, `CROUCH`, `WALK`, and `IDLE` are mutually exclusive ground-based states selected based on input.

4 Resource Budgets

4.1 Memory and Storage

1. **On-Chip FPGA Resources:** The current custom hardware is limited to the audio peripheral, which uses FIFOs, clock division logic, and codec initialization state machines. This is lightweight compared with a full framebuffer design.
2. **HPS Memory:** The software side uses normal HPS memory for game state, player states, rendering buffers, and input parsing. The HPS has sufficient memory for the current gameplay implementation.

3. **Future VGA Resource Considerations:** A full framebuffer in FPGA on-chip memory would be expensive, so the preferred future design is on-the-fly rendering driven by MMIO registers rather than storing entire frames in hardware memory.

4.2 Bandwidth and Computational Constraints

- **Game Logic:** The gameplay logic is lightweight enough to run at approximately 60 FPS on the HPS.
- **Input Bandwidth:** Keyboard input bandwidth is small and not a bottleneck.
- **Audio MMIO Traffic:** Audio sample writes are manageable through the lightweight HPS-to-FPGA bridge.
- **Future Video Path:** The main future performance challenge is implementing a real-time FPGA VGA pipeline, not the current gameplay computation itself.

5 Hardware/Software Interface

5.1 Audio MMIO Registers

The implemented audio peripheral is memory-mapped at `0xFF203040`. Its main registers are:

- `0x00`: control / status
- `0x04`: FIFO space
- `0x08`: left-channel sample data
- `0x0C`: right-channel sample data

These registers allow software to check codec status, monitor FIFO capacity, and write sample words.

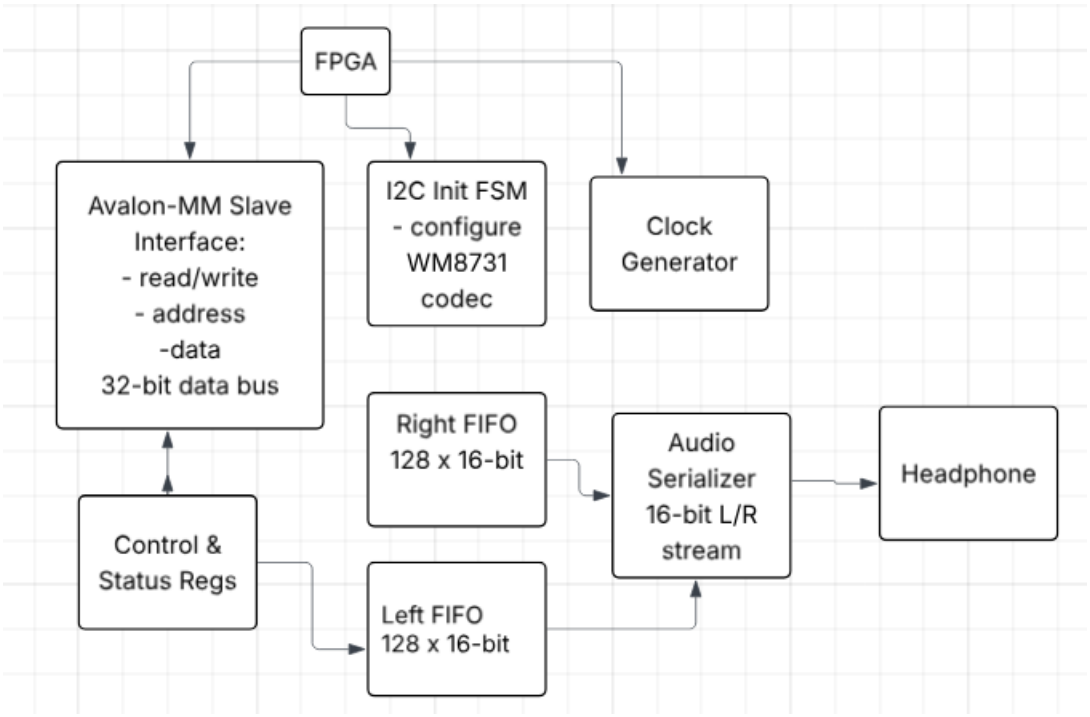

Figure 2: Hardware general architecture

5.2 Fighter MMIO Contract

The project also defines a future `fighter_mmio` contract for VGA rendering. This contract is already stable on the software side and exports 22 32-bit words.

The data includes:

- global game state
- player coordinates
- player visual states
- HP values
- round timer
- facing direction
- attack command information
- state-frame counters
- event flags
- combat result fields

This interface is intentionally richer than a minimal coordinate-only interface so that a future FPGA VGA renderer can draw not only the fighters, but also HUD elements, effects, and debug overlays.

5.3 Write Sequence from Software

The current software-side sequence is:

1. collect player input
2. update the internal game state
3. encode visual/gameplay state into MMIO-friendly fields
4. write audio data or future rendering data to hardware-facing interfaces

This separation ensures that the FPGA does not need to interpret high-level game rules.

5.4 FPGA Decoder Logic (future video path)

In the future VGA design, the FPGA video block will continuously read the exported fighter MMIO state and determine:

- where each fighter should be drawn
- which animation/state should be displayed
- how health bars and timer should be rendered
- whether special event indicators or UI overlays should be shown

6 Testing and Validation

6.1 Gameplay Testing

The current repository already includes automated testing for key gameplay behaviors. These tests validate:

- menu entry and round start
- exiting back to menu
- KO transitions
- game-over restart conditions
- timeout resolution
- hit-confirm behavior
- block-stun behavior
- facing reversal after cross-up situations
- overlap resolution between grounded players

6.2 Current Limitations

The project still has several limitations:

- VGA rendering is not yet implemented in FPGA hardware
- the current menu path is simplified
- audio is not enabled by default
- there is no dedicated Linux kernel driver yet

7 Conclusion

This project already implements a playable two-player fighting game prototype on the DE1-SoC platform. The current system includes working HPS-side gameplay logic, software rendering, dual-keyboard input, automated tests, and an FPGA-backed audio peripheral.

The next major step is to connect the existing `fighter_mmio` interface to a custom FPGA VGA renderer. That extension would preserve the already working gameplay logic while moving video output into hardware, resulting in a cleaner and more complete hardware/software co-design.