# Design Document for Spaceship Defender Game

CSEE 4840 Embedded System Design

Stephen A. Edwards (se2007) Spring 2025

Mingzhi Li (ml5160), Noah Hartzfeld (nah2178), Hiroki Endo (he2305),

Jingyi Lai (jl6932), Zhengtao Hu (zh2651)

April 18, 2025

# Contents

# 1 Introduction

Welcome to the high-flying world of our Spaceship Defender Game! This project creates an engaging space shooter experience utilizing FPGA hardware acceleration for smooth gameplay and graphics rendering.

In this project, our team will create a single-player arcade-style game based on Galaxian. The user will control a single spaceship at the bottom of the screen, firing towards rows of enemies above. They must avoid enemies firing back at them, as well as dive bombing towards the ship. With limited lives and a mission to defend against the alien invasion, only the most skilled pilot will emerge triumphant!

Our system combines hardware acceleration with software game logic, running on a DE1-SoC platform with VGA output and controller input support. The user will interface with the software through an SNES gamepad, communicating over its own USB protocol. The software will communicate with the FPGA using a software kernel driver, while the FPGA will handle displaying the graphics for the game, as well as the audio. The 60 FPS gameplay provides responsive controls and fluid animation, delivering an authentic arcade experience reminiscent of classic space shooter games.

# 2 System Block Diagram

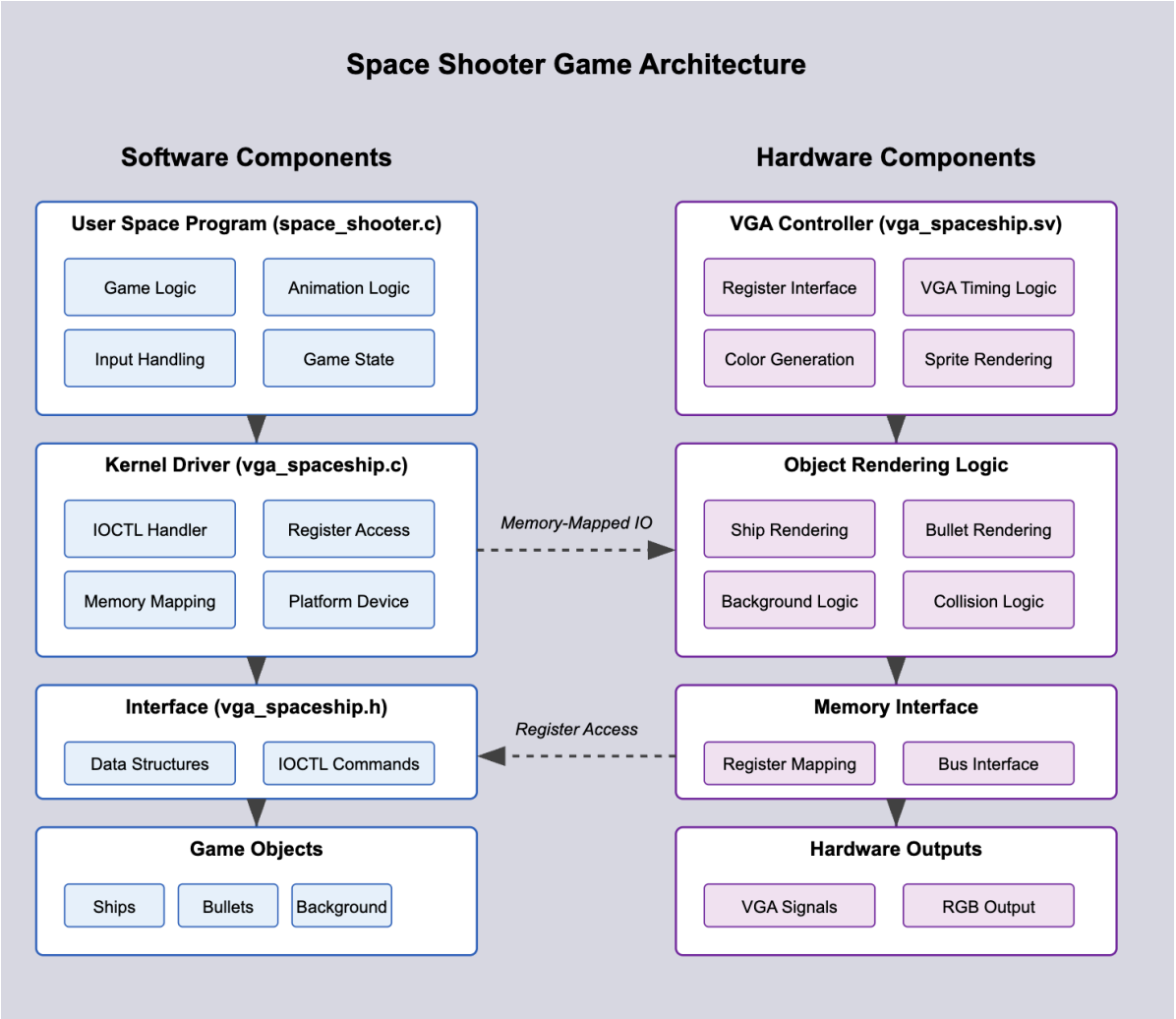


Figure 1: Modular Decomposition of the Space Shooter Game System Architecture

This diagram provides a comprehensive decomposition of the Space Shooter game system, clearly delineating the software and hardware components that, together, create a complete gaming experience. The left side illustrates the software stack, consisting of multiple hierarchical layers that handle game logic, state management, and the interface to hardware. The right side depicts the hardware implementation that manages image rendering, memory, and physical display output.

The software section showcases the user-space program (space_shooter.c) responsible for game mechanics. This includes gamestate tracking and modifying, the requisite animation decisions, and input processing. Not shown in the diagram is the program to find and open a connected controller so that inputs can be processed (controller.c).

Below this, the kernel driver (vga_spaceship.c) bridges user applications and hardware, implementing IOCTL handlers, memory mapping registers to the hardware, and defining access functions for these registers. The interface layer (vga_spaceship.h) defines data structures and IOCTL commands that standardize communication between software layers. At the bottom, the game objects represent just some of the conceptual entities within the game state, manipulated by the software.

The hardware section illustrates the FPGA implementation, beginning with the VGA controller (vga_spaceship.sv), which handles register interfacing, timing logic, color generation, and sprite rendering. The object rendering logic translates the abstract game state into visible entities on screen. The memory interface provides the necessary address mapping and bus connectivity, while the hardware outputs section manages the physical VGA signals and RGB color output.

# 3 Algorithms

Hardware:

The main hardware algorithm is the logic required to generate and display graphics on the VGA monitor. This will be achieved using the TMS9918 architecture with sprite and tile graphics.

Our background will be stationary and solid black with different patterns of stars, and will be displayed using a grid of preloaded background tiles stored in a background table, alongside their memory addresses, in RAM. Similarly, we will have a sprites table in RAM to easily store and select sprites for displaying. All the images, including spaceship, enemies, bullets, and explosion effects, are treated as sprites placed on a static tiled background.

A line-by-line scan-based rendering logic will be used. For each pixel, the system will compute the current coordinates and determine whether any sprite or tile needs to be displayed at that location. If so, the address is calculated and the corresponding color value is read from ROM.

All .png images are preprocessed into .mif files and compiled into on-chip ROMs. Sprite attributes such as position and type are controlled through memory-mapped registers.

Inputs are synced with v_sync to avoid image tearing, as no positional changes will be made during drawing. Sprites will be rendered on top of background tiles when they overlap.

Software:

**Basic game logic**

This is a single-player game. The player will control a spaceship object on the screen using an SNES Gamepad controller. The objective of the player is to survive as long as possible; shoot as many enemies before being hit by enemy bullets or directly by an approaching enemy, and running out of lives. Each player will start with 5 lives. Once they are hit, they lose a life and gain a short period of invincibility. Once one player loses all five lives, the game is over.

The start position of the spaceship is centered in the bottom half of the screen, and the controllable region for the player is bounded only by the limits of the screen. The plane has free movement across the screen, up, down, left, right, and diagonally. The player can press a button to fire shots vertically up towards enemies.

Each round will contain a set number of enemies, aligned in rows, beginning at the top of the screen. As the ship moves back and forth, enemies in the same vertical column as the ship will shoot bullets back down towards it.

If an enemy is struck by a ship's bullet it is killed and the bullet is terminated. If an enemy's bullet strikes the ship it loses a life. If any bullet reaches the screen boundry it is terminated. Both enemy and ship bullets will move in a straight line trajectory; moving until it hits an enemy or the ship, an opposition bullet, or reaches the screen boundary.

The player will be allowed a maximum of five bullets on the screen at a time, and will have a small refresh time between each bullet. A single bullet is allowed for each enemy, and only near enemies can fire.

As the round goes on individual enemies will leave their rows and dive towards the ship. If an enemy strikes the ship it will lose a life and the enemy is killed. If the enemy reaches the bottom of the screen without hitting the ship, it will respawn at the top of the screen and return to its position in row. Once all enemies have been destroyed the round is complete, and new and more enemies will be spawned in.

**Power ups**

We plan to implement a progressive difficulty mechanic for this game. When the player starts a new game, there will be a round counter. Enemies will grow progressively stronger, with faster bullet speed, firing rate, etc every new round. In return, the player can also get reinforcements by eliminating enemies. Every 10-20 enemies killed will drop a power-up such as faster firing rate, a brief protective shield, or a life bonus.

# 4 Resource Budgets

Table 1: Resource Budgets

| **Category** | **Item** | **Size** | |
|---|---|---|---|
| Graphics | Spaceship Player | 32*32*24 = 24576 bits = 3 KB | |
| | Five Enemies | 16*16*5*24 = 30720bits | 、 |
| | Bullet*3 | 8*8*24*3 = 4608bits | |
| | Explosion Effect | 32*32*24 = 24576 bits = 3 KB | |
| Audio | 0.5s Explode (8kHz, 16bit) | 0.5×8000×16 = 64000 bits = 8 KB | |
| | 0.25s Laser (8kHz, 16bit) | 0.25*8000*16 = 32000 bits = 4 KB | |
| | 3.0s Engine loop (8kHz, 16bit) | 3×8000×16 = 384000 bits = 48 KB | |
| | 0.75s Powerup (8kHz, 16bit) | 0.75×8000×16 = 96000 bits = 12 KB | |
| | 1.5s Gameover (8kHz, 16bit) | 1.5×8000×16 = 192000 bits = 24 KB | |

# 5 The Hardware/Software Interface

Table 2: Address distribution

| Address Range | Description | Details |
|---|---|---|
| 0-2 | Background color | RGB components (8-bit each) |
| 3-6 | Ship position | X(11-bit), Y(10-bit) coordinates |
| 7-26 | Player bullets | 5 bullets × 4 registers each |
| 27 | Player bullets status | 5-bit bitmap for active bullets |
| 28-35 | Player bullets status | 2 enemies × 4 registers each |
| 36 | Enemy status | 2-bit bitmap for active enemies |
| 37-60 | Enemy bullets | 6 bullets × 4 registers each |
| 61 | Enemy bullets status | 6-bit bitmap for active bullets |
| 62-63 | Image position | X coordinate (partial) |

## 5.1 Controller

We will be using an iNNEXT SNES gamepad controller to interact with the game. The controller is connected to the SoC via a USB port.


Figure 3: The iNNEXT SNES gamepad controller

The software will communicate with the controller using the libusb library to capture input and pass it to the software in a readable format. The user can control the movement of the ship with the controller's D-pad, and fire bullets by pressing the Y-button or either of the bumpers.

## 5.2 Audio

On the DE1-SoC board, all game audio is routed through the on-board Wolfson WM8731 CODEC, which offers two 24-bit DAC channels (and ADCs) plus a headphone amplifier, supporting sample rates from 8 kHz up to 96 kHz. The WM8731 is configured over an I2C control bus (shared between HPS and FPGA via the on-board multiplexer) by the Altera "Audio and Video Config" IP core.

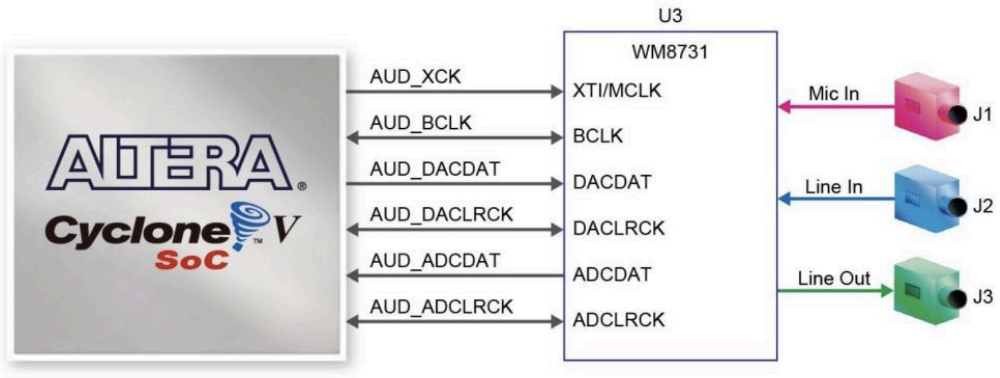


Figure 4: Interface between FPGA and Audio CODEC

Our spaceship shooter will feature five effects: laser blasts, explosions, a looping engine hum, power-up chimes, and a game-over jingle. They are all stored as 16-bit mono PCM at 8 kHz in left-justified format, so both channels play the same data.