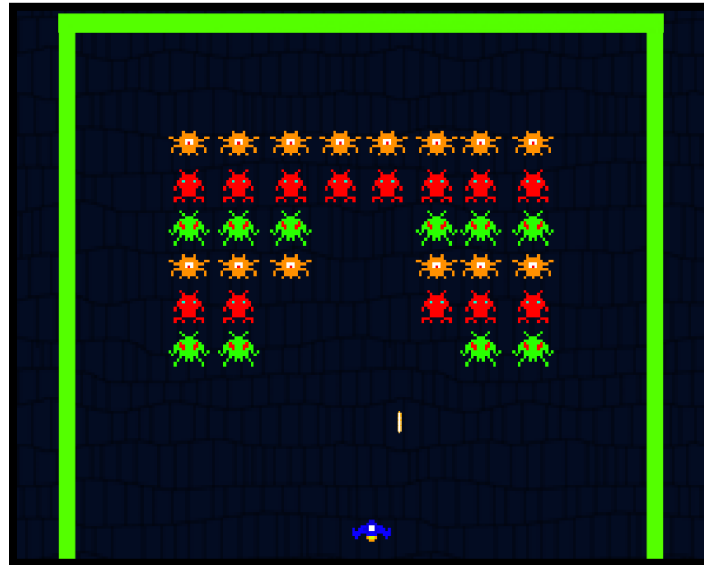


Design Document for CSEE 4840 Embedded System Design: Space Invaders Revamp



Alan Hwang (awh2135)
Zachary Burpee (zcb2110)
Mili Sehgal (ms6657)

Spring 2023

Contents

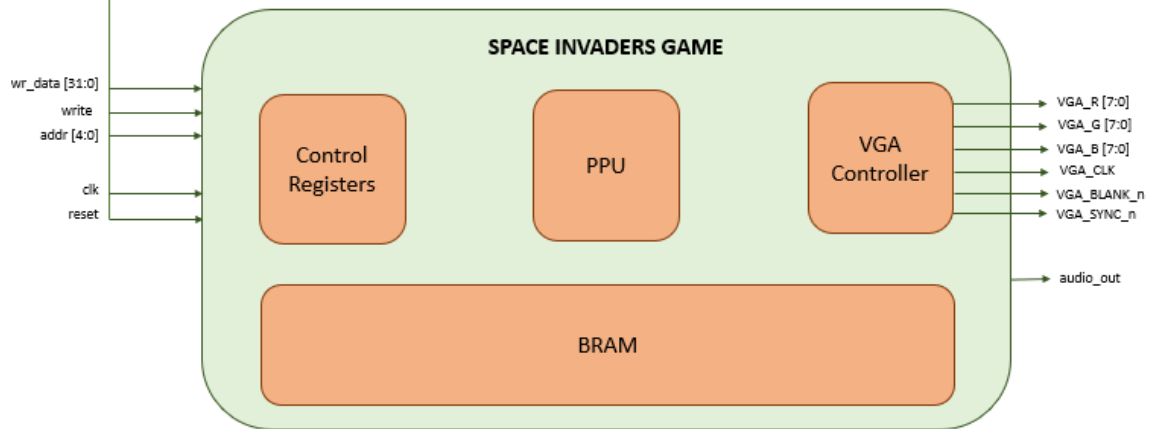
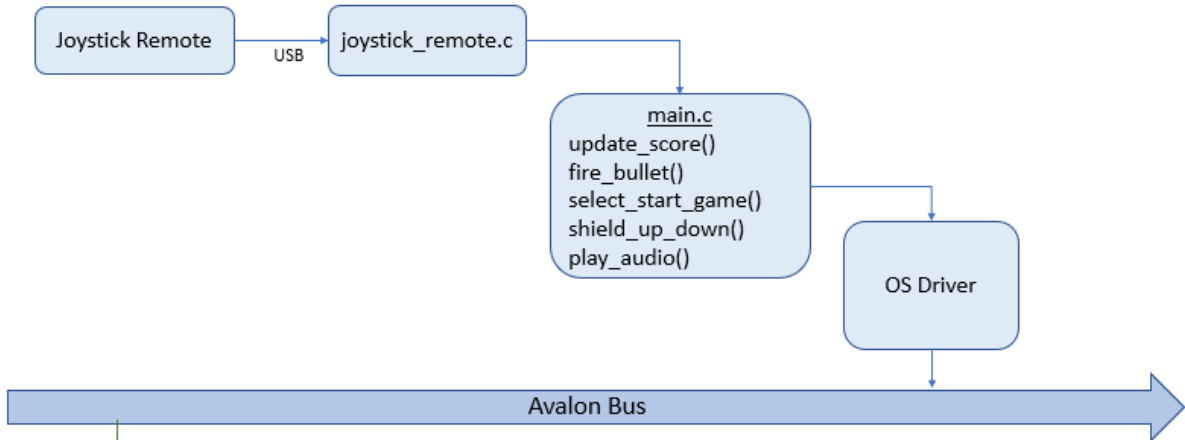
1. Introduction
2. System Block Diagrams
3. Algorithms
4. Memory Budget
5. Hardware Requirements
6. Software Requirements

Introduction

We will recreate the classic “Space Invaders” game from the 1970s. A VGA will display the background and continually add space invaders along with a controllable player ship for defending the Earth. The user will control the position of the defending spaceship and fire at enemy ships using an external joystick controller. If the user is able to successfully eliminate all space invaders, they will proceed to the next level. We plan to incorporate different levels of difficulty to incorporate strategy (i.e. powerups, space invaders that move fast or have more health). We will be using sprite graphics to display our visuals.

System Block Diagram

Software



Hardware

Algorithms

The joystick peripheral must have communication algorithms that will relay important information for each button. The following functions will be implemented:

- `move_left()` → button movement will indicate left translation of pixels of player ship
- `move_right()` → button movement will indicate right translation of pixels of player ship
- `fire_bullet()` → button press will launch bullet pixels from player ship
- `shield()` → button hold will put up a shield against incoming bullets of player ship
- `start()` → start button will start game in beginning
- `select()` → select button will reset game after lives are terminated or game is won

The following state algorithms will keep track of the gameplay from the input states of the joystick. We need to focus on the essential states:

- Enemy ship state
- Player ship state
- Game interface (level, points, lives, audio)

Enemy Ships State - The enemy ships will be “invading” the Earth by slowly moving down toward the defending ship and firing periodic bullets too. During the process, they will be “bouncing” back and forth across the screen and turn directions each time the end ships hit the screen edge. A sample depiction of the ship organization is shown below.



The enemy ship locations will be sent to the hardware for display from their X and Y positions. The movement of the downward trend will speed up in accordance with the level the player has achieved. Once the lowest ship reaches the player or if all the enemy ships are eliminated, the game is over. Additionally, different levels of enemy ships can appear as the player progresses, which will require more shots to defeat the enemy ship. The enemy ship state will have to maintain these hitpoint values.

Player Ship State - The player ship will be “defending” the Earth by firing shots at the incoming enemy ships. The player ship will receive movement, firing queue, and shield commands from the joystick peripheral. The player ship can be harmed by the incoming bullets from enemy ships and a life will be taken from the game interface. The player ship does not have a limited number of bullets, but it does have a cooldown on how fast the player can fire. An integrated clock will need to be used to determine the cooldown of at least 0.1 seconds.

Game Interface - The game interface will keep track of the current level, current lives, current points, and signal different audio tracks when an event occurs. The levels will increase once all the enemy ships have been defeated. The lives will decrease after the player ship is hit with an

enemy bullet. The points will increase according to the enemy ship defeated by the player ship. The audio tracks will be determined by an event taking place - for example, an enemy is damaged, the game is over, the player takes damage, and background music. All explanations of levels, enemy ship hitpoints, and audio events are shown in the Resource Budget tables.

Pseudocode Functions & Structs:

```
struct player {
    int x,y;
    bool fire, shield;
    int lives;
    int points;
};

struct invader {
    int x,y;
    int alive;
    char direction; /* 'l'=left 'r'=right */
    int level;      // Level on screen as it descends
    int hitpoints;
};
```

```
// Set player settings
int UpdatePlayerSettings()

// Set enemy ship settings
int UpdateEnemySettings()

// Move algorithms
int MovePlayer()
int MoveEnemy()
int MoveBullets()
int RockHealth()

// Display the score, lives, level
int DisplayScore()

// Increase level
int LevelValue()

int main() {
    // Initialize structs and game settings
    struct player tank;
    struct alien aliens[30];
    struct shoot shot[3];
    unsigned int input, loops=0, i=0, j=0, currentshots=0, currentaliens=30;
    int random=0, score=0;







    /* Display game title,score header,options */
    move(0, (COLS/2)-9);
    addstr("--SPACE INVADERS--");
    move(0,1);
    addstr("SCORE: ");
    move(0, COLS-19);
    addstr("m = menu  q = quit");


    while(1) {
        UpdatePlayerSettings()
        UpdateEnemySettings()
        MovePlayer()
        MoveEnemy()
        MoveBullets()
        RockHealth()
        DisplayScore()
    }

    gameover(win);
    endwin();
}
```

Resource Budget

Graphics

Category	Image/File	Size	Variants	# of Bits
Enemy Ships		36 x 36	3 Different Ships - Level 1 Ship (10 points) - Level 2 Ship (20 points) - Level 3 Ship (40 points)	$(36 * 36) * 3 * 24 = 93,312$ bits
Player Ship		36 x 36	1	$(36 * 36) * 1 * 24 = 31,104$ bits
Rock Shields		36 x 12	1	$(36 * 12) * 24 = 10,368$ bits
Bullets		12 x 12	2 - Green: Defender Bullet - Red: Enemy Bullet	$(12 * 12) * 2 * 24 = 6,912$ bits
Lives		18 x 18	1	$(18 * 18) * 1 * 24 = 7,776$ bits
Level		18 x 18	3	$(18 * 18) * 3 * 24 = 23,328$ bits

Background		18 x 18	1	$(18 * 18) * 1 * 24 = 7,776$ bits
Total				180,576 bits

Audio

Category	Time (s)	Frequency (kHz)	# of Bits
Background Music	15.2	8	$121,593 * 16 = 1,945,488$ bits
Damage Enemy	0.23	8	$1,815 * 16 = 29,040$ bits
Enemy Destroyed	0.35	8	$2,869 * 16 = 45,904$ bits
Player Takes Damage	0.23	8	$1,815 * 16 = 29,040$ bits
Player Loses a Life	0.35	8	$2,869 * 16 = 45,904$ bits
Game Over	2	8	$16,200 * 16 = 259,200$ bits
Total			2,354,576 bits

Total number of bits required: **2,535,152**

The DE1-SoC board provides 4,450 Kbits. Our memory size is only 2,536 Kbits, so our initial design should fit well within the provided resources.

Hardware Requirements

a) DE1-SoC FPGA Board:

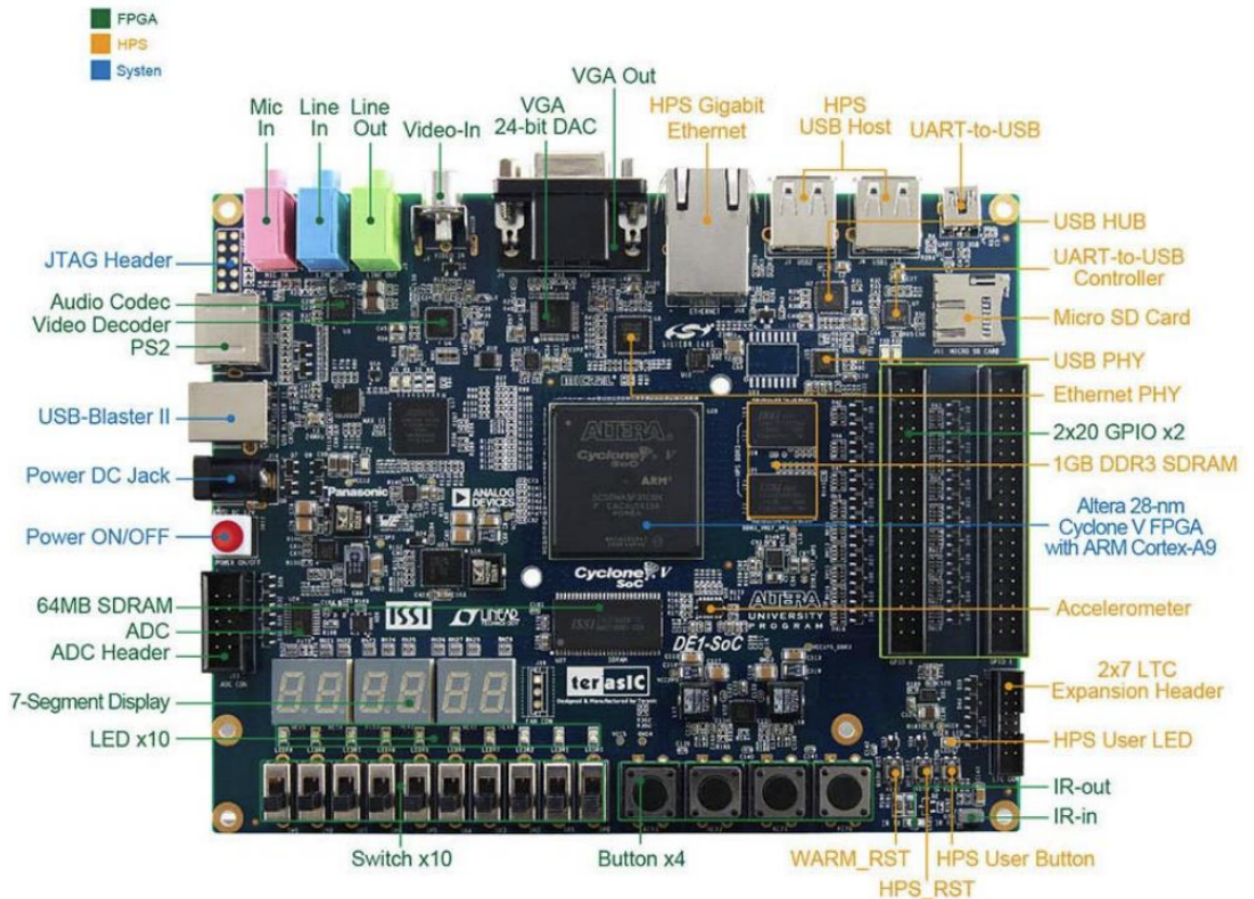


Fig: DE1-SoC Board Block Diagram

b) VGA Display:

Background layer: The background layer includes space/plain background that provide a setting for the game.

Foreground layer: The foreground layer could include game objects such as the player's ship, enemy ships, bullets, and other moving objects. This layer could be animated to show movement and changes in the game environment.

User interface layer: The user interface layer could include elements such as score counters, life indicators, and other graphical user interface (GUI) elements. This layer is designed to display information that helps the player keep track of the game status.

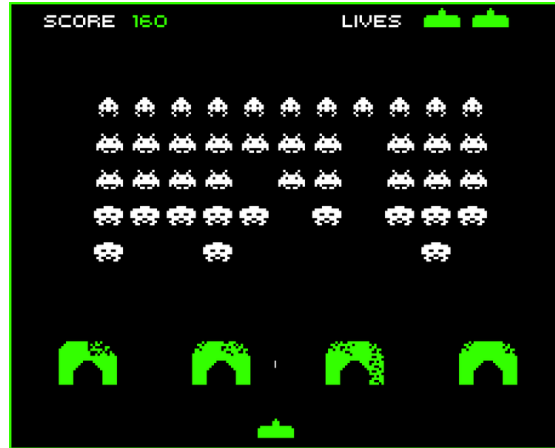


Fig: Sample Stills from the Game

c) **Joystick Control:** The Joystick is connected to the DE1-SoC Board using the USB Port. We can control the game using the following buttons on the Controller:

- Press Button-A for firing the Bullet
- Press Left Arrow to make spaceship move Left
- Press the Right Arrow button to make spaceship move Right
- Press Button-B for a protective Shield



Fig: Controller that will be used for the Game

d) Speakers: The speakers will be connected to the audio output port on the DE1-SoC board. These are controlled by the pulse width modulation (PWM). We will configure the audio CODEC to receive and output the PWM audio signal by setting the sampling rate, bit depth, and other parameters using the serial control interface.

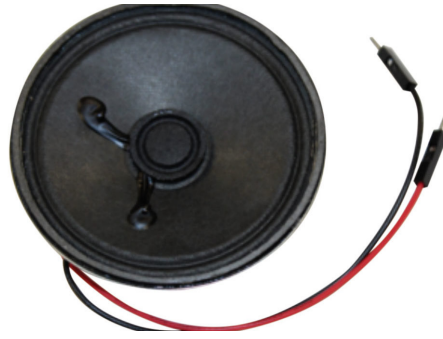


Fig: Speaker for Audio Output

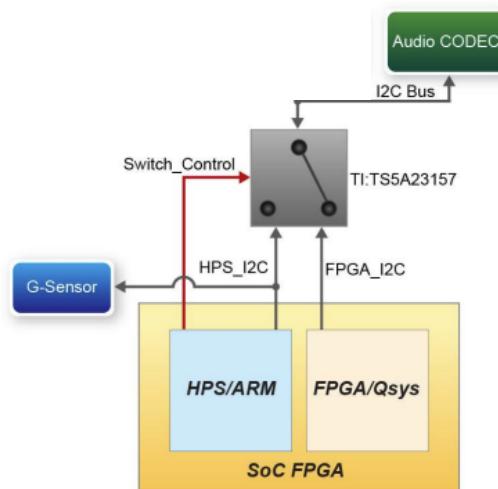
Software Requirements

We require the following software tools for implementing the Space Invaders game on FPGA DEI-SoC:

a) HDL Programming Language: To create the game logic, we will use hardware description language (HDL). HDL will allow us to describe the digital circuitry of our game, including the processing units, memory elements, and input/output interfaces.

b) Audio Controller:

The Audio Controller module in the FPGA design controls the playback of the digital audio samples, allowing the game to produce realistic sound effects and music as and when required. First, convert the audio files to .v files, then use Audio CODEC WM8731 to output the sounds to the speaker after configuring it using I2C multiplexer.



Control Mechanism for I2C Multiplexer