ECE 385 Final Project: Loose Cannon

Tiger Hou | linyih2 | github.com/TigerHou2

May 11, 2021

Contents

1 Overview

As my final project for ECE 385, I created a Team Fortress 2 themed, artillery-based fighting game that was inspired by Worm Clan Wars / Tank Wars. The project was run on a DE10-Lite FPGA and displayed on a VGA monitor, while controls were issued through a USB keyboard. Core game mechanics were implemented in hardware using SystemVerilog. SPI communication protocols for the USB keyboard and in-game AI were implemented in software using C, which was run on a NIOS II/e CPU instantiated via Platform Designer.

2 Playing the Game

2.1 Objective

Loose Cannon can either be played as a single-player or two-player game. Players each control a character in either red or blue uniforms, with the goal of eliminating the other character by reducing their health to zero using bombs before their opponent can do the same. Pressing `KEY1` toggles between single-player/two-player modes. [Single-player demo]

2.2 Movement

Movement is controlled by three keys — jump, move left, and move right. For Player 1 (P1), these keys are `W`, `A`, and `D`, while for Player 2 (P2) these keys are `I`, `J`, and `L`.

2.3 Shooting

Players can launch bombs that fly along parabolic trajectories. The angle and power of the bomb can be adjusted for a better shot. For P1, the angle can be rotated counterclockwise with `Q` and clockwise with `E`. Power is controlled by holding down the "shoot" key, which is `S`. Holding the button longer results in a faster shot. For P2, the controls are `U`, `O`, and `K`, respectively. Each player can launch one bomb at a time, with the next one being available as soon as the current one explodes.

2.4 Damage

Bombs explode upon collision with either the opposing player or terrain. The explosion has a radius of 30 pixels and deals damage depending on how close the opponent is. At point-blank range, the bomb deals 55 damage. At minimum, the bomb deals 25 damage. Half of the damage dealt is recoverable at a rate of 2.5 HP/s. Health yet to be recovered is removed when additional damage is taken. Players start with 100 HP.

2.5 Terrain

Terrain can be deformed by bomb explosions. The bomb removes terrain within a 19-pixel radius upon explosion. Terrain generation is randomized at the beginning of each game. Using switches 0-9 on the DE10-Lite, different terrain seeds can be selected. Once a new seed is chosen, press `KEY0` on the FPGA to load the new terrain. Note that this will reset the game as well.

3 Game Mechanics

A schematic of the final project is shown in Figure 1. Finer details of the connections, as well as some less relevant, smaller modules, can be seen in the more detailed RTL diagram shown in Figure 2.

In this section, the hardware elements that impact gameplay will be described in greater detail. In the next section, graphics elements used to generate and render the game will be discussed.

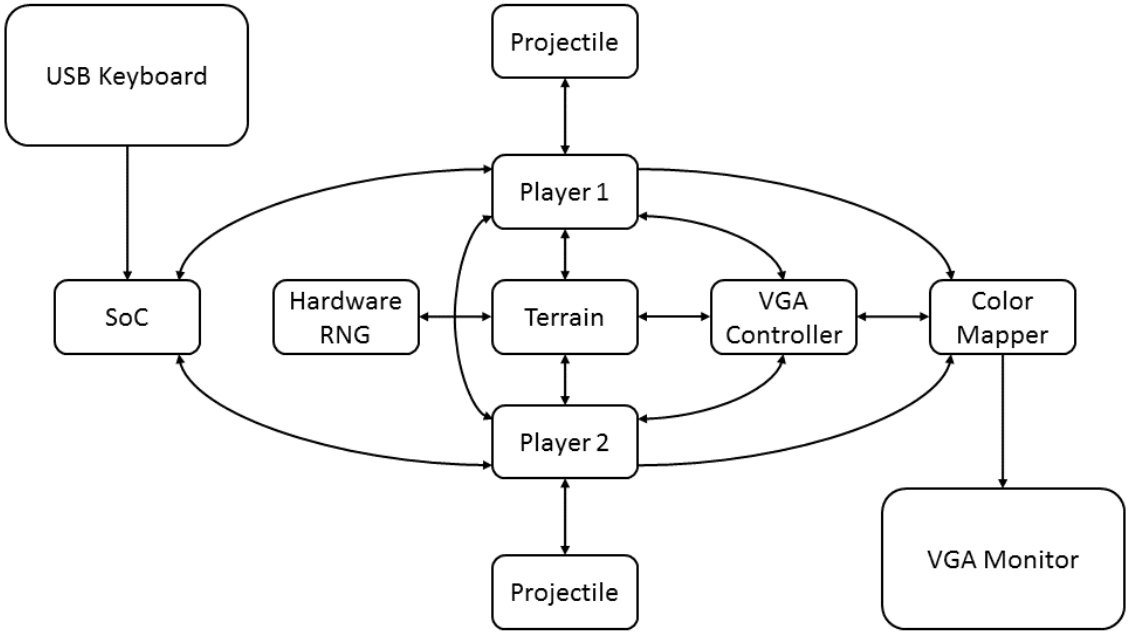

Figure 1: Schematic of the final project design.


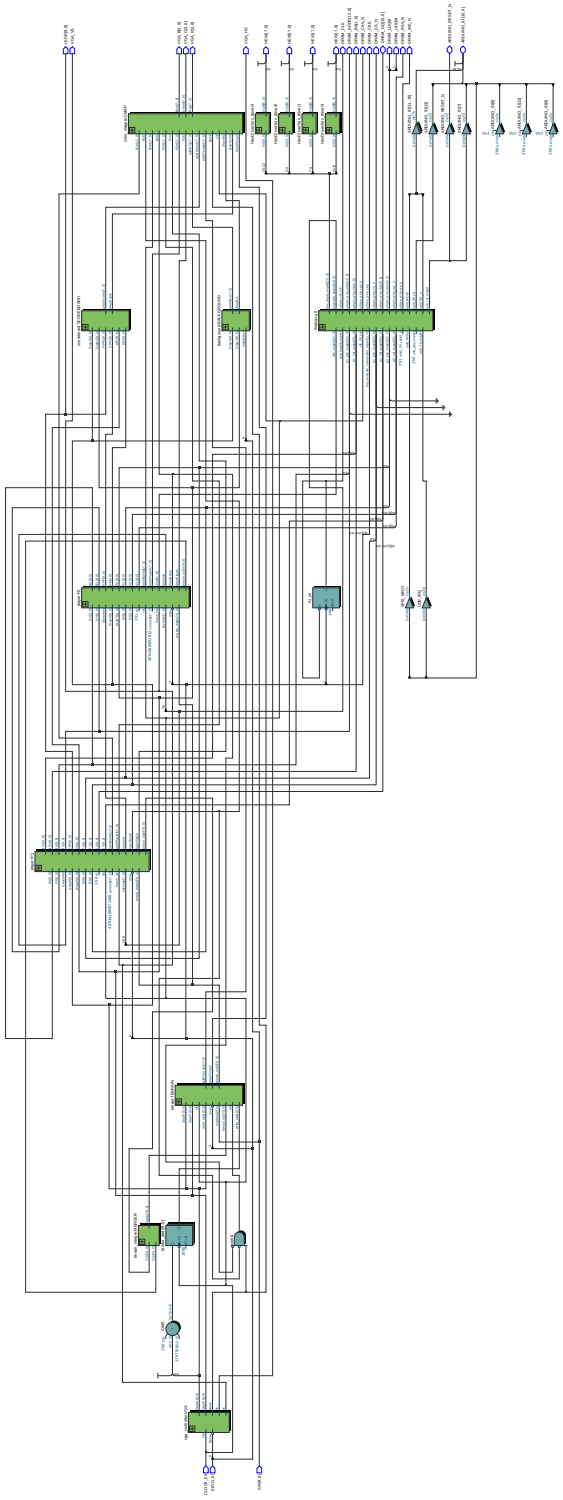

Figure 2: Top-level RTL view of the final project design.

3.1 Player

The `player` module handles the movements, actions, damage tracking, and sprite reading for a character. To facilitate these capabilities, several sub-modules are instantiated in addition to the code within the `player` module.

3.1.1 Core Code

Upon a reset, the character's position is set to a predetermined position on screen, and the character's velocity is set to zero. Health is set to the maximum of 100.

The `player` module runs using the frame clock, which is approximately 60 Hz. Each frame, the character's new position and velocity are determined through a series of modifications:

a. Gravity is applied by adding 1 to the vertical component of the velocity. Since this would produce too much acceleration, the effective gravity is reduced by only adding gravity once every six frames.

b. Collision detection is performed. Landing on terrain would immediately zero out all velocity. Colliding with terrain in any other direction would cancel out velocity in that direction. If a character becomes stuck in terrain due to moving too quickly in one frame, then they are ejected by reducing their velocity to 1 and reversing the velocity direction, bouncing the character out of the terrain.

c. Player input is applied. Movement in the x-direction (left, right) add or reduce velocity by 1 and could by applied once every six frames. Movement in the y-direction (up only) is interpreted as a jump by adding 4 to the velocity, but could only be applied once every 32 frames.

d. Terminal velocity is enforced. Velocity in the x- and y-direction are both capped at ±7 pixels per frame.

e. Screen bouncing is performed. Bouncing against any screen border would cause that velocity component to be reduced to 1 in the opposing direction. At the same time, the position component would be bounded to the screen border to ensure the character would not glitch out of bounds.

f. The character's position is updated by adding the new velocity to the current position.

The `player` module also handles aiming, shooting, and damage tracking. Aiming is done by tracking player inputs which can adjust either the launch angle or launch power of the bomb. There are 9 possible angles and 8 possible powers, for a total of 72 possible

trajectories, which can cover the vast majority of the map when used in conjunction with some character movement. The angle and power values are then translated into bomb velocity by the bomb sub-module, which is described in Subsection 3.2. Shooting is controlled by the same input as power — when the input is released, the bomb is launched.

Damage is tracked by subtracting an amount of health proportional to the character's distance from the explosion. Half of the damage could be recovered over time, though any damage not yet recovered is lost when more damage is taken.

3.1.2 Sub-Modules

The `player` module contains the following sub-modules: `terrain_collider`, `bomb`, and `mask`. The `terrain_collider` handles collisions with the terrain by checking whether the character's bounding box is on a pixel that has terrain. If this is true, then a flag in the corresponding direction is set to 1, indicating that a collision has occurred.

The `bomb` sub-module manages a bomb entity which is launched by the character. Due to its complexity, it is described separately in Subsection 3.2. The `mask` sub-module applies transparency to the character sprite. It is described in Subsection 4.2.

3.2 Bomb

The `bomb` module is instantiated by the `player` module, and manages the movement, interactions, and sprite reading for the bomb entity.

3.2.1 Core Code

Similar to the `player` module, the bomb is subject to gravity and terminal velocity constraints. Unlike the player, however, the bomb is spawned at the character's position when the power input is released. Upon a reset or explosion, the bomb is reset to a position outside of the boundaries of the map, and gravity is not applied until the bomb is launched again. The initial x- and y-velocities of the bomb are determined via a lookup table.

3.2.2 Sub-Modules

The `bomb` module contains a `terrain_collider` sub-module and a `player_collider` sub-module. The former is already described in Subsection 3.1.2, while the latter computes whether the opponent is within the explosion radius of the bomb. A collision indicated by either module would trigger the explosion event. This causes the bomb to check each pixel of the terrain to determine whether that pixel should be removed by the explosion.

3.3 Terrain

Terrain is stored using on-chip RAM and generated using pseudo-random number generators. The `terrain` module runs at approximately 50 MHz, and sends an entire column of terrain (1 pixel wide) to the top-level at once.

3.3.1 Core Code

The `terrain` module provides terrain data to the graphics and player modules, and also receives modified terrain data from bomb explosions. Terrain is regenerated upon a reset, using the value of switches 0-9 at the time of reset as the new seed for the pseudo-random number generator.

Terrain generation is perturbed by pseudo-random noise generated by the `PRNG` module described in Subsection 3.3.2. Noise is implemented with the following formula:

$$E_{k+1} = \frac{7}{8}E_k + \frac{1}{8}n_k - 59$$

where E_k is the noise at the k^{th} step and n_k is the pseudo-random number at the k^{th} step. The terrain height is then implemented as a random-walk:

$$h_{k+1} = h_k + E_k$$

3.3.2 Pseudo-Random Number Generators

Pseudo-random number generation is achieved using linear feedback shift registers (LFSRs) and cellular automata shift registers (CASRs). A CASR operates on each bit of the state by performing an XOR operation between its two neighboring bits. An LFSR uses a linear function of its constituent bits (such as XOR) to determine the next state. An example of an LFSR is shown in Figure 3.

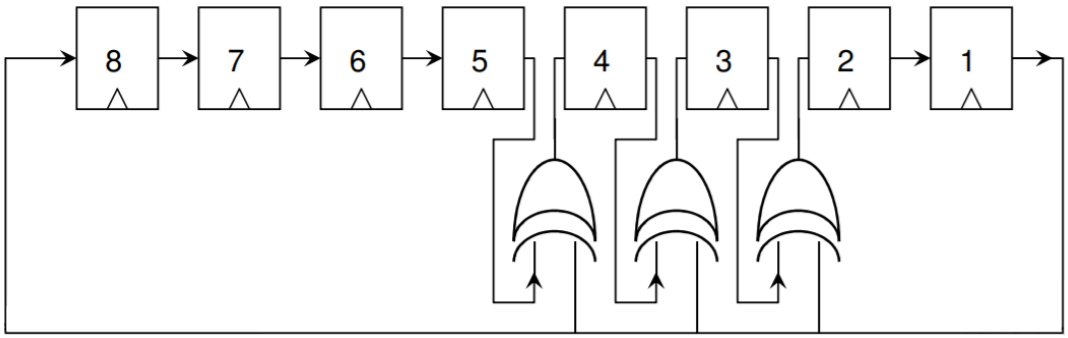

Figure 3: Example of an LFSR in Galois form [1].

Within the PRNG sub-module, two LFSRs and one CASR are implemented. This is done to increase the path length of the pseudo-random number generation process to prevent early repetitions. The outputs of the three shift registers are XOR'ed together to generate a 10-bit pseudo-random number which is used to create randomness in the terrain generation process.

3.3.3 On-Chip RAM

The SRAM sub-module stores terrain data in the on-chip RAM, specifically the M9K blocks, as a 480×640 array. Each clock cycle, data from an entire column of 480 pixels is read. The column is specified by the horizontal position of the virtual electron beam, which is detailed in Subsection 4.1. At the same time, an entire column of updated terrain data is written into the terrain RAM, which includes modifications to the terrain due to gameplay. Since it takes one clock cycle to compute the terrain modifications, the write address is always one column behind the read address.

4 Graphics

In this section, graphics modules and tools used to create sprites for the game are explained. Note that some paragraphs of this section are copied over from the Lab 6.2 report.

4.1 VGA Controller

The `VGA_controller` module manages the synchronization pulses and other timing signals used to properly display graphics to the VGA monitor. In addition, it also generates the horizontal and vertical coordinates of the pixel to be rendered on each positive clock edge. More specifically, the module serves as a virtual electron beam that draws each pixel of a 640 x 480 display in row-major order. By halving a 50 MHz clock input to 25 MHz and sweeping over 800 x 525 pixels (this includes the front and back porches as well as the sync pulses), an overall framerate of approximately 60 Hz is achieved. The module also sends out a blanking signal, which ensures that pixels are drawn only when the virtual electron gun is on the display, not when it is pointed at the front or back porches, nor the sync pulses.

4.2 Sprite Drawing

All sprites for the game are stored in the on-chip memory. Since the sprites are only read from and never written to, the memory was implemented as ROM. To fit all sprites into the limited number of M9K blocks, a palette-based approached is taken to reduce the number of bits required to store the data. A total of 32 colors were provided, which allowed the palette to fit to 5 bits per pixel.

Given the current coordinates of the electron beam, the position of a sprite, and the size of the sprite, the palette index at the corresponding ROM address could be accessed. The `color_mapper` module could then use a lookup table to find the corresponding color and send that RGB signal to the VGA monitor.

4.3 Color Mapper

The `player`, `bomb`, `terrain`, and `background` modules all receive the electron beam coordinates, and each return a ROM address to the `Color_Mapper` module. In addition, they also send a draw enable signal. The `Color_Mapper` module selects a single address among the enabled sprites based on priority, in the order of bomb →player →terrain →background →blanking signal. A blanking signal is used when the electron beam is off-screen — it is set to black and is necessary for the proper operation of the display.

After the address is selected and the palette index is read from RAM, a lookup table is used to determine the RGB values corresponding to the palette index. Finally, the RGB values are sent to the VGA monitor to be displayed.

4.4 Graphics Preparation

The images used in the game needed to be processed before being stored in RAM. In addition to using a palette-based approach, the size of the background images also needed to be scaled down to be able to fit two different backgrounds into the RAM. In addition, since the sprites for the game were found online, the number of colors needed to be reduced to fit the palette limit of 32.

The MATLAB function `rgb2ind` was used to reduce an image to a set number of colors. Dithering was applied to one of the backgrounds to improve the overall appearance. Finally, a python script from "Rishi's ECE 385 Helper Tools" provided on the course wiki page was used to convert the processed images into a format that could be read by the `readmemb` function in Verilog.

5 System-on-Chip

The system-on-chip (SoC) portion of the project was largely carried over from Lab 6.2. The SoC was used to instantiate a NIOS II/e processor and provide communication between the USB keyboard and the top-level design. In addition, new PIO modules were added to transfer data from the top-level, containing player and bomb states, to the C code running on the NIOS II for the single-player A.I.

5.1 IP Core Instances and Descriptions

Table 1: List of Platform Designer IP Core instances and purposes.

clk_50	Clock, generates a clock signal for all IP cores except the SDRAM.
nios2	NIOS II/e processor, processes data and issue commands to all IP cores.
oc_mem	On-chip memory, placeholder.
sdram	SDRAM, stores the software to be run on the NIOS II/e processor.
sdram_pll	Phase-lock loop (PLL) module, supplies a skewed clock to the SDRAM.
sysid	System-ID checker, ensure compatibility between software and hardware.
leds_pio	PIO module, sends data to the FPGA LEDs.
jtag_uart	JTAG module, allows r/w to the CPU through the USB via interrupts.
keycode	PIO module, receives keycodes from the USB keyboard.
usb_irq	PIO modules to communicate with the MAX3421E USB chipset.
usb_gpx	
usb_rst	
hex_pio	PIO module, sends data to the FPGA 7-segment displays.
key	PIO modules, receives keypresses from the FPGA buttons key0 and key1.
timer	Interval times, keeps track of USB timeouts.
spi0	SPI module, allows the NIOS II/e to communicate with the USB chipset.
p1_pos	PIO module, position of player 1.
p1_vel	PIO module, velocity of player 1.
p2_pos	PIO module, position of player 2.
p2_vel	PIO module, velocity of player 2.
b1_pos	PIO module, position of player 1's bomb.
b1_vel	PIO module, velocity of player 1's bomb.
aim	PIO module, player 2's aim angle and power.
aim_toggle	PIO module, toggles A.I. for player 2.

5.2 SoC Schematic View

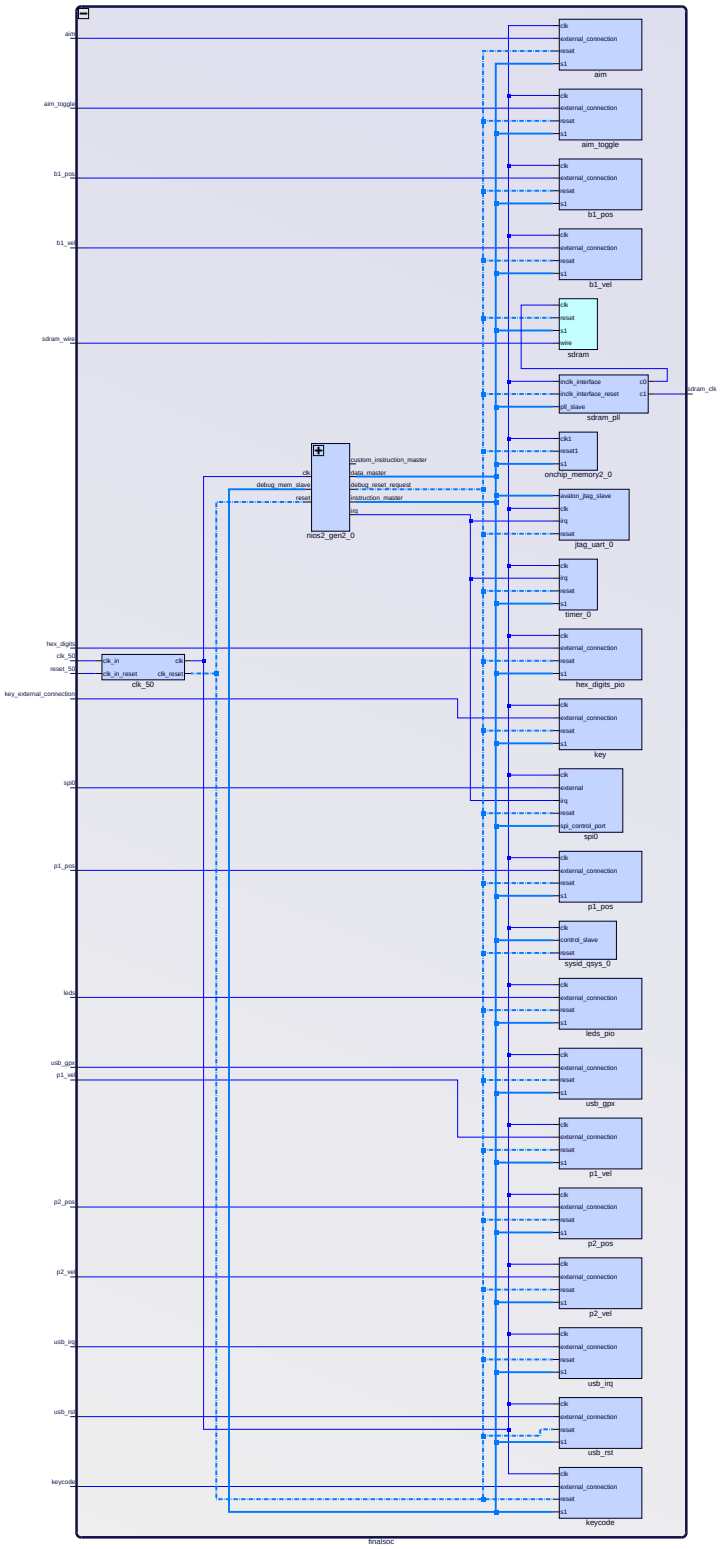

Figure 4: Schematic view of the SoC design for the final project.

6 Software

Software for the final project is heavily based on Lab 6.2, where the low-level SPI communication protocols with the USB keyboard were implemented. For Loose Cannon, modifications were made to detect keycodes within two sets of allowed keys — namely, keys corresponding to the controls of P1 and P2. Since the USB keyboard is capable of sending up to six keycodes at once through SPI, the first allowed key for both P1 and P2 are selected and sent through the `keycode` PIO module to the top-level.

6.1 A.I. Design for Single-Player NPC

In single-player mode, an A.I. takes control of P2. The A.I. is designed in C and run on the NIOS II/e. It takes the position and velocity of P1, P2, and incoming bombs as input, and produces a keycode as the output. As such, the A.I. has no advantage over the player in terms of information or control authority. Pseudo-code for the A.I. is shown below:

```
calculate closest approach of P1 bomb based on current velocity
if (close approach < threshold): jump

calculate distance from enemy
for (all combinations of power and angle)
    calculate bomb parabola
    if (intersect enemy y_position)
        calculate x_displacement
        if (x_displacement < best solution)
            update desired angle & power
            update best solution

if (best solution < tolerance)
    if angle incorrect: adjust aim
    if power incorrect: adjust power
    else: release bomb
else
    if landed: jump
    else: move to reduce x_displacement
```

A demo video of the gameplay against the A.I. is provided here. The banded artifacts come from recording a VGA screen; this is not an issue when viewed directly.

7 Summary

A screenshot of the game is shown in Figure 5, and a video demo of the game can be viewed here. A summary of the design statistics of the final project is shown in Table 2.

Loose Cannon was a fun final project to work on for the class. It called upon many skills I learned throughout the course of ECE 385, and I was also able to pick up some new knowledge related to memory, graphics, pseudo-random number generation, etc. To me, the most challenging element of this project was terrain generation/modification — many hours were spent trying to figure out the correct timing needed to properly store terrain data. Other time-consuming parts included finding ways to maximize efficiency in image storage, and of course, tweaking the physics of the game itself.

In the end, I think I created a fun game that I was proud of. Me and my roommates played a couple of scrimmages and our match got more intense than it had any right to be :). I became more confident in thinking on the hardware level and writing SystemVerilog, not to mention debugging skills in general.

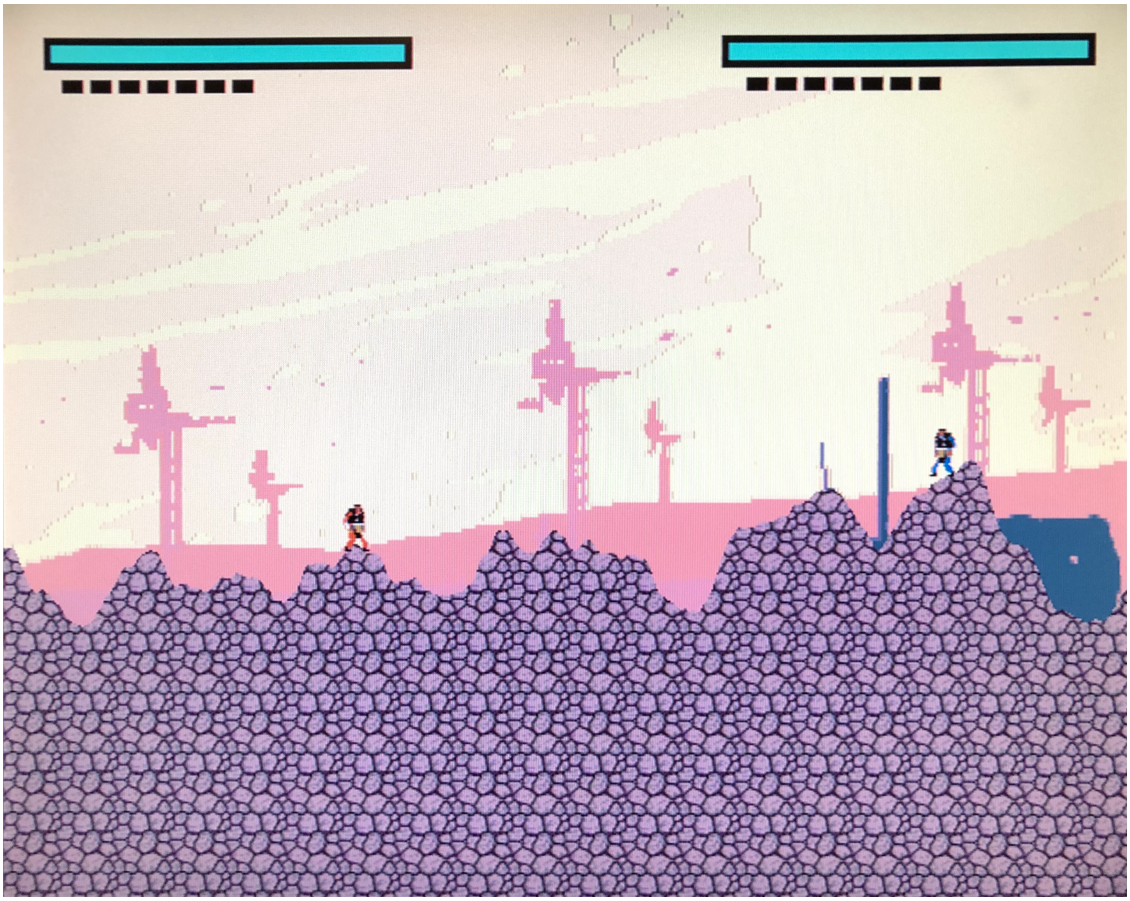

Figure 5: Loose Cannon screenshot.

Table 2: Final project design statistics.

LUT	9,690
DSP	16
BRAM (bits)	1,456,128
Flip-Flop	4,716
Frequency (MHz)	42.67
Static Power (mW)	97.08
Dynamic Power (mW)	180.48
Total Power (mW)	299.03

Appendix A Description of .sv Modules

```
Module:  vga_controller
Inputs:                 Clk, Reset,
Outputs: logic          hs, vs, pixel_clk, blank, sync,
                [9:0]   DrawX, DrawY
Description:
    Counts in row-major order for an 800 x 525 array of pixels. Generates a
    pixel_clk signal half the speed of the incoming Clk signal. Sets hs high at
    pixels 656-752 of each row. Sets vs high signal at rows 490-491. Sets blank high
     when pixel > 639 or row > 479. The signal sync is not used. Outputs the column
    (X) and row (Y) coordinates as DrawX and DrawY.
Purpose:
    This module is used to generate horizontal and vertical sync, as well as
    blanking signals for the 640 x 480 VGA display.
```

```
Module:  color_mapper
Inputs:                 clk, blank,
                        P1D, P2D, B1D, B2D, drawBG, drawTerrain, drawScore,
                [17:0]  P1A, P2A, B1A, B2A, addrBG, addrTerrain, addrScore,
Outputs: logic  [7:0]   Red, Green, Blue
Description:
    This module implements combinatorial logic to select a ROM address from which a
    palette index is retrieved. The palette index is then mapped to RGB colors. The
    module outputs RGB channel colors for each pixel specified by DrawX and DrawY.
Purpose:
    This module handles all drawing to the VGA display.
```


```
Module:  spriteROM
Inputs:                 clock,
               [17:0]   address,
Outputs: logic [4:0]    q
Description:
    1-port ROM with 5-bit output.
Purpose:
    This module stores image color palette data, provided in .txt files, to the on-
    chip memory. All sprites used in the game are in this module.
```

```
Module:  maskROM
Inputs:                 clock,
               [9:0]    addr1, addr2,
Outputs: logic          out1, out2
Description:
    2-port, 1-clock ROM with single-bit outputs.
Purpose:
    This module stores masks used to create transparency effects for character
    sprites. An output of 1 means to draw the pixel, while an output of 0 means to
    ignore the pixel and draw the next layer instead.
```

```
Module:  background
Inputs:                 mapSelect,
               [9:0]    DrawX, DrawY,
Outputs:                drawBG,
               [17:0]   addrBG
Description:
    This module uses combinatory logic to generate the ROM address for the color of
    the pixel at the current draw position. It also accepts two ROM offset values to
     draw two different images.
Purpose:
    This module generates the ROM address for the background pixels.
```

```
Module:  player
Inputs:                 clk, reset, frame_clk, ID, Dboomed,
               [479:0]  terrain_data,
               [7:0]    keycode,
               [9:0]    DrawX, DrawY, DX, DY, EX, EY,
               [47:0]   controls,
```

```
Outputs:                boomed,
                [9:0]   PX, PY, VX, VY, BX, BY, BVX, BVY, HP, HPP,
                [31:0]  aim,
                        drawPlayer, drawBomb,
                [17:0]  addrPlayer, addrBomb,
                [479:0] terrain_out
Description:
    This module performs the functions described in Subsection 3.1. The inputs
    include the position and velocity of the enemy and bomb, as well as terrain,
    control inputs, and some timing and clock wires. Outputs include the character
    and bomb's position, velocity, health, as well as pixel color address and
    modified terrain data.
Purpose:
    This module handles all features related to the player characters.
```

```
Module:  bomb
Inputs:                 clk, reset, frame_clk, launch,
                [9:0]   launchX, launchY, DrawX, DrawY, EX, EY, boomRadius,
                [3:0]   angle,
                [2:0]   power,
                [479:0] terrain_data,
Outputs:        [9:0]   X, Y, VX, VY,
                        drawBomb, boomed,
                [17:0]  addrBomb,
                [479:0] terrain_out
Description:
    This module performs the functions described in Subsection 3.2. The inputs
    include the bomb's initial position, the enemy's position, the explosion radius,
     the launch angle and power, as well as terrain data and some timing and clock
    wires. Outputs include the bomb's position and velocity, whether the bomb has
    exploded, the pixel color address, and modified terrain data.
Purpose:
    This module handles all features related to the bomb.
```

```
Module:  terrain_collider
Inputs:                 clk, reset,
                [479:0] terrain_data,
                [9:0]   X, Y, DrawX, D, U, L, R,
Outputs:                DD, UU, LL, RR
Description:
```

```
    This module takes vertical slices of terrain data and its horizontal position to
     determine whether an entity (character or bomb) at position [X,Y] has collided
    with the terrain. The collision boundaries in each direction are specified
    separately, and four collision indicators are generated as output.
Purpose:
    This module handles all collisions with terrain.
```

```
Module:  player_collider
Inputs:         [9:0]   X, Y, PX, PY, radius,
Outputs:                collided
Description:
    This module calculates whether a bomb's position [X,Y], is within [radius]
    distance of the enemy player's position [PX,PY].
Purpose:
    This module computes collision between a bomb and the enemy player.
```

```
Module:  terrain
Inputs:  logic          clk, we, reset,
         logic  [9:0]   DrawX, DrawY, read_addr, write_addr, rngSeed,
         logic  [479:0] terrain_in,
Outputs: logic  [9:0]   terrain_height,
         logic          drawTerrain
         logic  [17:0]  addrTerrain,
         logic  [479:0] terrain_out,
Description:
    This module performs the functions described in Subsection 3.3. Terrain in
    regenerated using a pseudo-random number generator upon reset, and is otherwise
    modified based on gameplay.
Purpose:
    This module handles all features related to the terrain.
```

```
Module:  SRAM
Inputs:                 clk, we,
                [9:0]   read_addr, write_addr,
                [479:0] data,
Outputs:        [479:0] q
Description:
    1-port RAM with data width = 480 and 640 addresses.
Purpose:
    This module stores terrain data.
```

```
Module:  PRNG
Inputs:                 Clk, Reset,
               [9:0]    Seed,
Outputs:       [9:0]    Out
Description:
    This module implements two LFSRs and one CASR described in Subsection 3.3.2. It
    takes a 10-bit seed and outputs a 10-bit pseudo-random number each clock cycle.
Purpose:
    This module is the pseudo-random number generator used for terrain generation.
```

```
Module:  scoreboard
Inputs:        [9:0]    DrawX, DrawY, HP1, HPP1, HP2, HPP2,
Outputs:                drawScore,
               [17:0]   addrScore
Description:
    This module draws health bars at fixed locations on-screen. The length of the
    health bars are determined by the status of the characters. The module outputs
    pixel color data to the color mapper module.
Purpose:
    This module controls the display of health bars for both players.
```

```
Module:  FinalProject
Inputs:                 CLOCK_50
               [1:0]    KEY
               [9:0]    SW
Outputs:       [7:0]    HEX0, HEX1, HEX3, HEX4,
               [12:0]   DRAM_ADDR
               [1:0]    DRAM_BA
                        DRAM_CLK, DRAM_CKE, DRAM_LDQM, DRAM_UDQM,
                        DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N
                        VGA_HS, VGA_VS
               [3:0]    VGA_R, VGA_G, VGA_B
In/Out:        [15:0]   DRAM_DQ, ARDUINO_IO
                        ARDUINO_RESET_N
Description:
    This module instantiates player, terrain, and graphics modules used for the game
    . It also instantiates the SoC to establish connection with the NIOS II/e.
Purpose:
    This is the top-level module for the final project.
```

References

[1] P. Schaumont, "Lecture 6: A random number generator in verilog." http://rdsl.csit-sun.pub.ro/docs/PROIECTARE%20cu%20FPGA%20CURS/lecture6[1].pdf Accessed: 5-10-2021.