ECE 385

Fall 2016

Final Project Report

Galaxian Space Shooter Game

Michael Brady & Utkarsh Pandey
NetID: mwbrady2, upandey3
Section: AB5

Table of Contents

1 Introduction

This project aims to emulate the operation of the classic Namco arcade game Galaxian using sprites from the updated version of the game, Galaga. The game is implemented through hardware on an FPGA with the use of a SoC processor to add a USB keyboard for commands. The gameplay is displayed on a VGA monitor in the original 224 x 288 pixel resolution. The basis of implementing the game through hardware are the sprites and the color mapper. In the project, the sprites are all stored in on- chip memory (OCM) due to their relatively small size. OCM was chosen because it is the fastest and the most convenient memory to access of the available onboard memories for our purpose, and because storage space was not a limiting factor. The color mapper takes these sprites and draws them according to the game state described in hardware.

2 Galaxian Gameplay Description and Instructions

The basic gameplay of our version of Galaxian is very similar to the original. Gameplay begins at the start screen which displays the title and makers of the game.

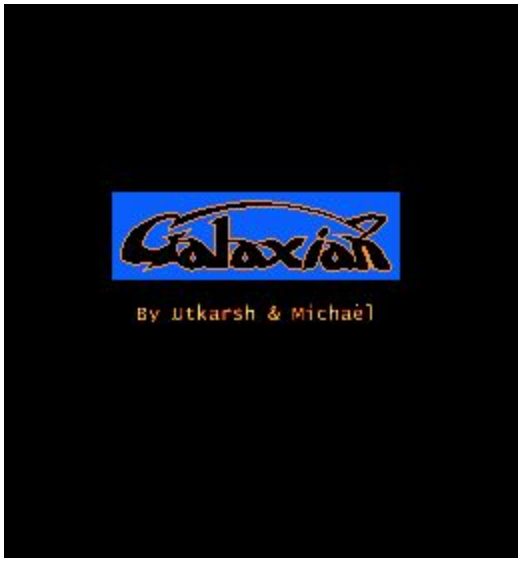

Figure 1. Start Screen

By pressing “enter” on the keyboard, the game advances to the gameplay and the player’s ship as well as all of the enemies appear.

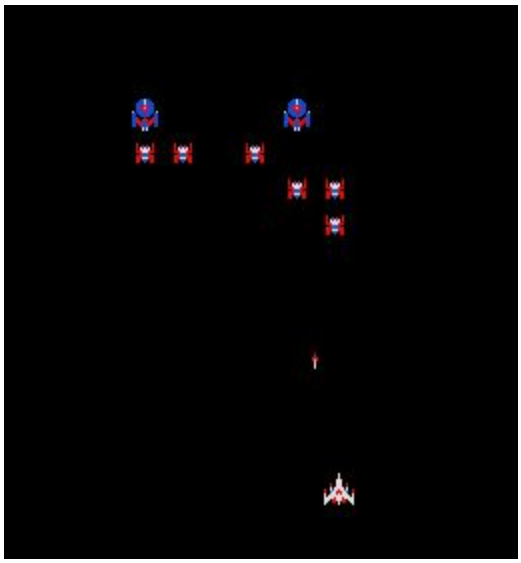

Figure 2. Gameplay Screen

The player can now move left and right using the arrow keys and shoot with the space bar. Two key codes can be simultaneously pressed if the player is trying to move and shoot at the same time. The player’s spaceship is stationary if there are no input commands, but enemies will move back and forth across the screen and appear to flap their “wings”. The objective is to shoot missiles from one’s own spaceship to hit the enemies at the top of the screen. When one of the missiles hits an enemy, both the missile and the enemy will disappear. Once all of the enemies are destroyed, the player wins and the game is over. The game can be lost though if the player is hit by an enemy’s missile. All the 24 enemies can randomly shoot out missiles downwards. If the player is hit by an enemy missile, the ship will explode. In either case, if the player takes out all of the enemies, or if the spaceship is destroyed, the game goes to the game- over screen where the player can choose to play again.

Figure 3. Game Over Screen

3 Description of Circuit and Modules

Our hardware circuit is programmed in SystemVerilog HDL and consists of .sv files that define all the modules that define the behavior of the entities in the game and the game-play itself.

3.1 Lab8.sv

This is the top-level module of our program. It declares the inputs and the outputs to our hardware program. It also instantiates and connects all the relevant modules such as ship, enemy, missile, gamestate and explosion, thereby facilitating the communication among the modules.

3.2 gamestate.sv

3.2.1 Gamestate description

The gamestate module is a basic state machine that transitions between three states: Start, level_1, and Game_Over. These states control the game state and the game goes from the start screen to the play screen and then to the game-over screen. Pressing “enter” on the keyboard will cause the game to transition from Start to level_1. In level_1, all enemies and the player appear. This is the main state for gameplay. If the module receives a signal telling it that the player has been hit by an enemy or that all enemies have been destroyed, it will transition to the last state, Game_Over. This sends out a signal to the color mapper for the Game Over screen to be drawn. The game can reset/go back to level_1 by pressing enter. This state machines runs on the frame clock which is the vertical sync signal from VGA_controller.

3.2.2 Gamestate state transition diagram

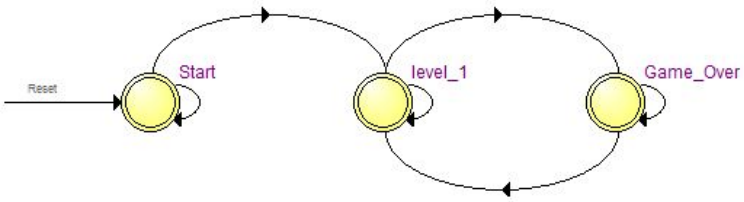

Figure 4. Game state transition diagram

3.2.3 Gamestate bench test simulation

Figure 5 displays the annotated game state transition waveform for a gameplay in which all enemies are destroyed.

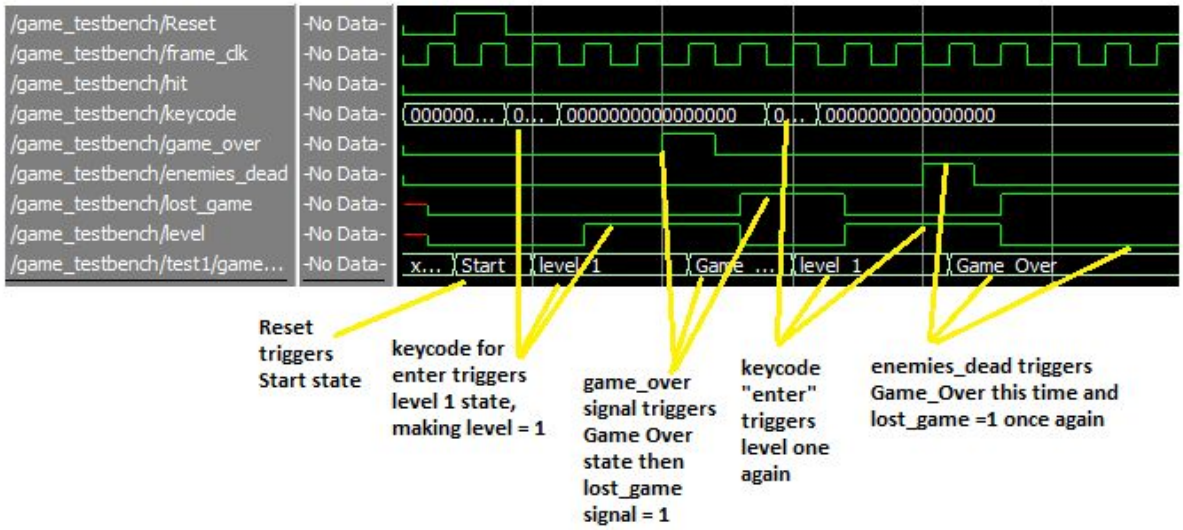

Figure 5. Gamestate testbench annotated simulation

3.3 ship.sv

The ship module specifies the operation of the player's ship which is controlled by keyboard commands. It outputs x and y coordinates of the ship and a signal to let the color mapper know if the ship is on or off. The module specifies a starting position for when the ship is spawned and updates position with movement of the ship every frame. The module also has parameters for a min and max x-position. If the ship exceeds these positions on the screen, it will not be able to move any further. If it is the minimum (left) position it will only be able to move right and vice versa. At reset, the ship's state signal is set to 0 and if enter is hit (also signifying beginning of gameplay) the signal will become 1, letting the color mapper to draw it.

3.4 missile.sv

3.4.1 Missile Description

The missile module is used to make the missile that the player shoots at enemies. It is different from the missile that is sent by enemies. The module outputs x and y coordinates of the missile and an on/off signal to let color mapper know if it should be drawn. Each frame clock cycle the module checks if it has made a hit, if it has gone off screen or if the fire key has been pressed. If it has gone off screen, it resets and it is off and when the fire key is pressed, it will turn on. When turned on, the starting position is set to the tip of the player's ship so that it appears that the missile is coming from the ship. Motion is set in the - y direction and each frame position is updated with previous position + movement.

3.4.2 Collision Detection

The module also takes in a signal to let it know if it has collided with an enemy. Color mapper outputs this signal if at a positive clock edge any enemy pixel is on as well as the missile. This would mean that part of the missile and part of the enemy overlap and have thus collided. When missile receives this signal it transitions between states from Hit_Off to Hit_On, which turns the missile off and resets it back to the start position (at the tip off the player's ship). At the next frame clock positive edge, the state becomes Trans, which is a transition state and automatically goes back to Hit_Off after another cycle. Both Hit_On and Trans set a signal which resets hit. This is output to the color mapper. Thus a missile will turn off when it hits enemy but is still able to be shot again because it resets as shown in Figure 6.

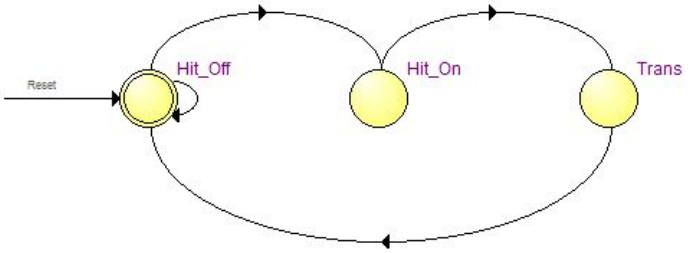

Figure 6. Hit state transition diagram

3.5 enemy.sv

The enemy module describes the behavior of the 24 enemy sprites. This includes their horizontal movement as well as the animation of flapping their wings. The module receives signals, level and lost_game and uses them to define whether the enemies should be on display. The module sends out 2 arrays for x and y positions for the 24 enemy sprites to the color mapper along with enemy_type, enemy_type2, enemy_type3. To disable drawing of the enemies, the enemy module sends enemy_type as 0. The on-screen enemies can be of three different types from a choice of 10 different sprites. The enemy module specifies the type of the 3 enemies using enemy_type, enemy_type2, enemy_type3. Each type of sprite has two different images which are alternately selected after each clock divider cycle. This emulates the flapping of the wings for the enemies.

The clock dividers are used on the frame clock in order to animate the enemies flapping and moving horizontally with the appropriate speed. Upon reset, the enemies are assigned their initial positions. But after that, the motion in the x coordinate is changed per the clock divider cycle. This creates the horizontal movement for the sprites and the updated positions are sent to the color mapper in order to draw the enemies according to those positions. In “start” state and in the “game-over” state, the enemies are disabled. Figure 7 displays the state transitions diagrams for selecting the different types of the same sprite in order to perform the animation for flapping of wings.

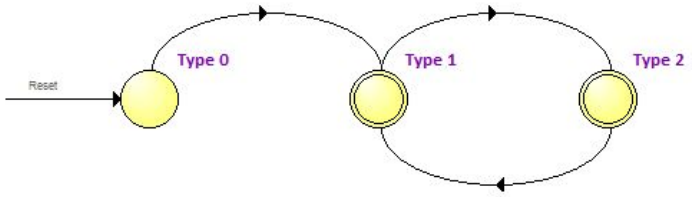

Figure 7. Enemy type state transition diagram

3.6 enemy_missile.sv

The enemy_missile module describes the behavior of the missile shot by the 24 enemy sprites downwards. The missile is shot randomly by one of the 24 enemies present at a time. The enemy_missile module takes as inputs two arrays from the enemy module, one for x coordinates and the other for y coordinates for the 24 enemies at each frame_clk. This module also takes as inputs an array called “present” from the color mapper which specifies whether an enemy is still alive. The module uses a linear feedback shift register to generate a random value between 0 to 32, out of which 24 values are mapped to the sprites. So if the register value maps to a sprite, then its x and y coordinates are then assigned to the enemy missile’s x and y start positions with the correct offset. Once that happens, the enemy_missile is transitioned from rest state to launch state and it shoots out with a positive y-step (y-direction). If the missile is not in the start position, it keeps going until it goes beyond screen dimensions, in which case it goes back to the rest state. If the mapped enemy is not present, it sets x and y positions of the enemy missile to 1, in which case, the enemy_missile stays in the rest state.

3.7 explosion.sv

The explosion module contains a state machine that dictates to the color mapper to draw 4 different explosion frames to create an animation of the player ship exploding. The “Start” state sends out 4 explosion outputs which tell the color mapper that the 4 explosions are off. When an enemy missile hits the player ship, the ship ceases to exist, but the an explosion also needs to occur. At this moment the color mapper sends out an “explode” signal to the explosion module to activate the explosion state machine. When the input “explode” is received, the state machine moves to the “explode0” state which sends out the signal to the color mapper to draw the first explosion image, and then the state machine moves to “explode1” and so on. The 4 explode states enable drawing of the 4 images at consecutive frame clock cycles to create an animation. After

“explode3”, the state machine moves to the “Game_Over” state which sends out a signal to to the gamestate module to indicate that the current game is over.

In order to ensure that the animation is not too fast to see, we determined through trial and error that the state machine should change every 8 frame clock cycles. We employed clock dividers for this reason. The explosion is shown below in Figure X.

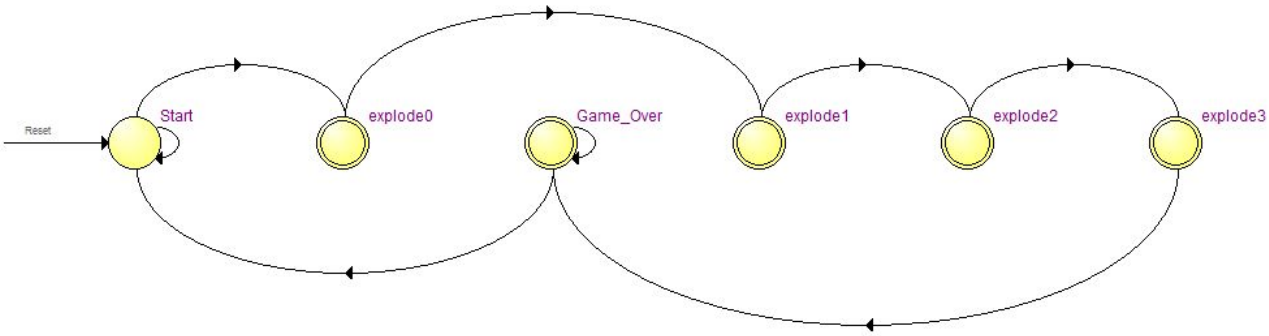

Figure 8. State machine for animating explosions for player ship

3.8 color_mapper.sv

The color mapper has a number of functions, the primary one being drawing the sprites on screen. All the sprites to be displayed ranging from enemy spaceships to logos for the start-screen and the game-over screen are inputs to the color mapper. The sprites are input through 4-bit arrays of appropriate dimensions. The 4 bits are used to map the 16 colors used in the game. The information related to the sprite objects, enemy, ship and missile are input to the color mapper such as the x- and y-coordinates of the object as well the enable signals for that object. TheDrawX and DrawY signals from the VGA controller are also input to the color mapper indicating the coordinates of the pixel being drawn. If the DrawX and DrawY point to a location where a certain sprite is located at that moment, it accesses the sprite array for that object for that coordinate and outputs the mapped-RGB values to the DAC controller enabling the VGA monitor to display that color at that pixel location.

3.8.1 Player_alive, hit & reset_hit, enemies_dead

Since the color mapper gets the coordinates of all the game objects and scans the frame pixels, it provides the opportunity to detect collisions between two objects. So for example, the “player_alive” signal is always set to be 1 until a collision between an enemy_missile and player spaceship occurs. To draw the player spaceship the “player_alive” signal has to be 1. So in case of such a collision, the color mapper stops drawing the player spaceship. Furthermore an “explode” signal is output at the same time to the explosion module to activate the explosion state machine which sends signals back to the color mapper to draw out explosion images instead of the spaceship.

Similar to “player_alive”, the color mapper has 24 logic signals for the 24 enemy sprites to define whether an enemy is alive or not. If all the enemies are dead, an “enemies_dead” is signal to output to game-state machine which then transitions to the “game over” state and sends out a signal to the color mapper to draw out the “Game Over” logo.

When player missile collides an enemy, the missile also has to stop from being drawn anymore. But the missile cannot be turned off in a similar approach as the enemy or the player ship as once the missile collides with an enemy, a new missile can be sent out again. The missile cannot simply be turned off as it needs to be reset to starting point (the coordinates of the player spaceship). To accomplish this, color mapper outputs a hit signal to the missile module which sends a reset_hit signal to the color mapper to ensure that the after the missile collision it is ready to be fired again from the player ship’s cannons.

3.8.2 Basic sprite drawing logic

To draw a sprite, it is required to know the x and y coordinates of the top-left

pixel of the sprite object, its x and y dimensions and DrawX and DrawY signals, which specify which pixel is being drawn on the screen currently. If the pixel being draw is within the rectangular area of the sprites' dimensions starting from the top-left pixel location of the sprite, the "sprite_on" signal will be set to the pixel value accessed by the sprite array for that location and the color mapped to the pixel value will be output as RGB signal from the color mapper.

The pixel access can be done using the following method
Ship_on = player[DrawY- ShipY][DrawX - ShipX]
where Ship_on is a 4-bit signal to contain the color key and ShipX and ShipY are coordinates of the top-left pixel location of the player ship.

3.8.3 Color key assignment and color key

As mentioned above, the sprite arrays contain 4-bit information(color keys) about the color of a certain pixel. Logic signals such as Ship_on and missile_on retrieve color keys from sprite array for a sprite object to be drawn. So if Ship_on is equal to 0, then ship is not supposed to be drawn at that location (outputs black color by default). If however, Ship_on is equal to a decimal value between 1 and 15, then a specific color is output by the color mapper. Figure 9 shows how the keys are mapped to the colors in SystemVerilog.

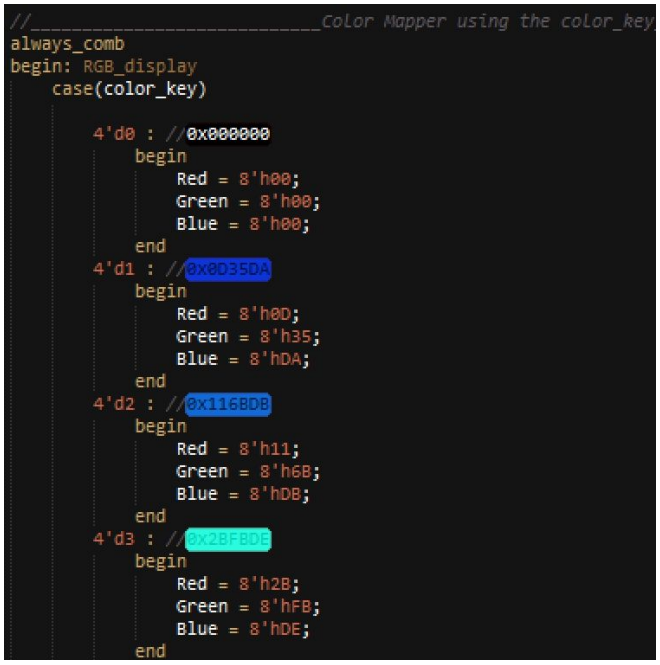

Figure 9. Example of how the color mapper maps 4-bit color keys

3.9 VGA_controller.sv

In the VGA_controller module, the horizontal and the vertical syncs are sent out to the VGA monitor and it also sends out the location of the current pixel being drawn to the color mapper. The VGA communicates through analog voltage signals using a DAC (digital-to-analog converter). The VGA (Video Graphics Array) is a matrix of 640 x 480 pixels. An “electron gun” displays the output from left to right in each row, from top to bottom. The screen is updated at 60Hz (60 times a second) and this communication is made possible through the VGA controller module. The hardware description modules and the Color Mapper modules specify what needs to be displayed at any moment using the inputs from the USB keyboard and the VGA controller respectively, as well as the information encoded into the game.

3.10 sprite_table.sv

The sprite table module is used initialize the sprite arrays in the on-chip memory and output them to the color mapper. The sprite arrays are two-dimensional arrays of dimensions, the height and the width of the sprite. The values correspond to a pixel position for that sprite. The arrays contain 4-bit values which are mapped to 16 different colors and the displaying of the correct RGB output based on these values is achieved by the color mapper. Figure 10 below demonstrates how the sprite table stores arrays of 4-bit values to map colors of the original image.

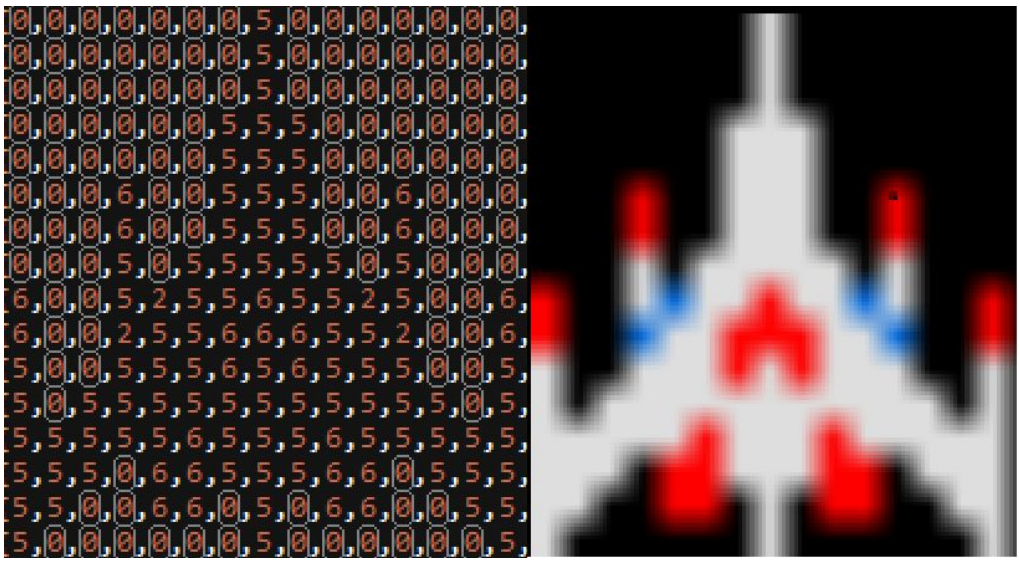

Figure 10. The Sprite Table array representing the player spaceship

3.11 NIOS II/e SOC

In order to interface a keyboard with the DE2 board, a USB connection is established by utilizing the CY7C6700 USB controller. The CY7C67200 is set up as a Host Controller which receives and processes the inputs from the keyboard. The USB communication works through a low speed transmission due to the small amount of data being passed at any given time and this information from the keyboard is sent upon a request from the Host Controller. The Host Controller does this by repeatedly polling input from the keyboard and sending interrupt signals to it when the information is needed in order to not block the functioning of

the CPU during polling buffer. The buttons pressed at that time send data(stored in a data structure called report descriptors) back to the Host Controller. The data is received by the Host Controller through the Human Port Interface (HPI) module that essentially directs the incoming and outgoing data(bidirectional) through four registers HPI_DATA, HPI_ADDRESS, HPI_STATUS, and HPI MAILBOX.

3.11.1 USB Integration

USBRead, USBWrite, IO_read and IO_write are all software functions which are used to communicate between hardware and software. IO_read and IO_write are both helper functions for USBRead and USBWrite that send signals to hardware out ports to read and write from and to memory. USBRead and USBWrite are used for access to the Cypress USB controller internal registers. Together they control both the memory and inputs and outputs that go to the hardware of the FPGA. In the USBRead function, IO_write is used to set the value of HPI_ADDR to the address that needs to be read from. IO_read is then used to return data from HPI_DATA by accessing the same address that was written to HPI_ADDR. USBWrite works very similarly, with the same first step, but instead of an IO_read, it uses IO_write again in order to set the value of HPI_DATA to the data being written to the controller.

4 Block Diagram

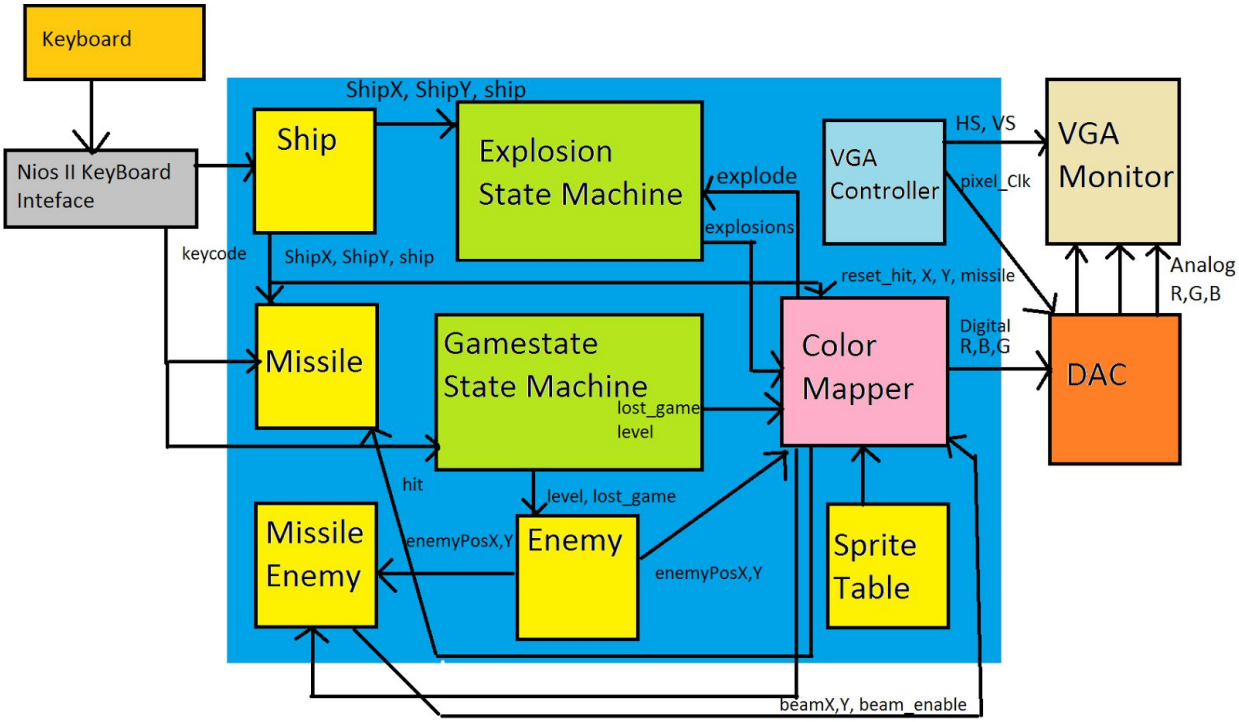

Figure 11. Top Level Block Diagram

5 Conclusion

5.1 Design Statistics

NIOS II/e with Galaxian	
LUTs	7,673
DSP Blocks	1
Memory (BRAM) (bits)	13,568
Flip-Flops	2,195
Frequency (MHz)	86.48

Static Power (mW)	102.33
Dyn. Power (mW)	2.7
Total Power (mW)	176.86

5.2 Conclusion

The idea for the project was inspired by the challenge to emulate an arcade game on hardware and in the end, the effort was successful. We were able to build the Galaxian game-play on hardware, emulate animations such as flapping of wings and explosions. We implemented all the basic functionality of the game to the point that more complications such as difficult levels or harder challenges can be easily added by using and building on our existing code. The challenges we faced during this process included failure to distinguish between behaviors based on the frame clock and those based on the actual clock. This often led to fruitless hours of debugging. But at the same time, using the debugging techniques acquired over the course of this semester, we were able to eventually solve those challenges, which in turn made us better hardware programmers.