ECE 385

Spring 2021

Final Project Report

Galaxian on the DE-10 Lite

Vikram Belthur & Santhosh Bomminani
netIDs: belthur2, snb3
Section ABF
TA: Abigail Wezelis

Table of Contents

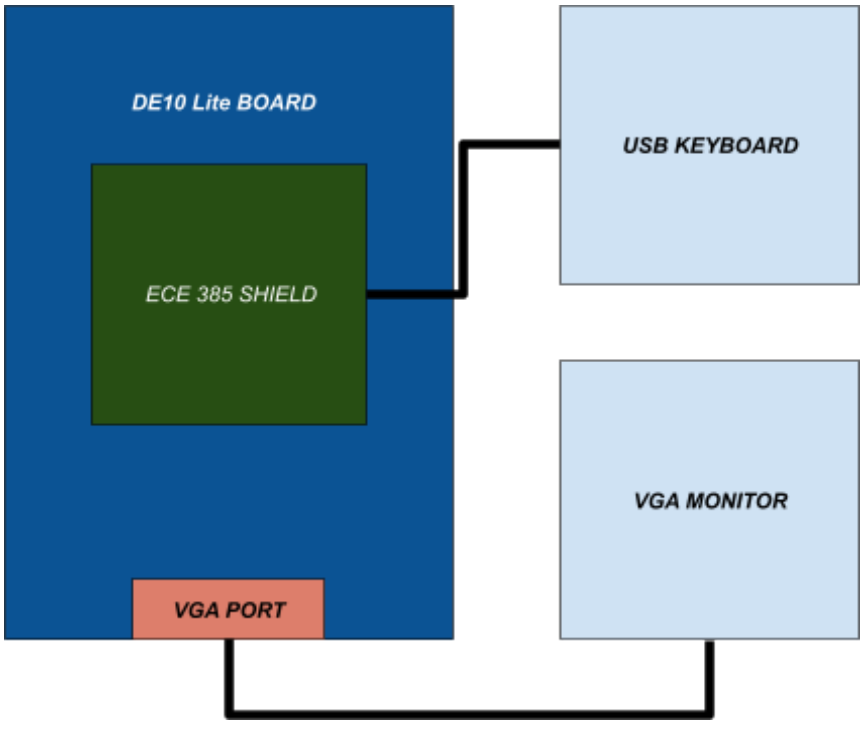
Introduction

For our ECE 385 final project, we opted to implement a FPGA-based game. The project, "Galaxian on the DE-10 Lite" is based on the classic arcade game Galaxian by Namco. Galaxian was released in 1979 and was the predecessor to the widely acclaimed game Galaga. Due to the similarities between these two games, this project has some visual elements that are reminiscent of Galaga and Galaxian. This game was implemented on the DE-10 Lite board by Terasic, and the FPGA on that board is the Altera MAX 10. The board is connected to a VGA monitor through its VGA port, and all visual elements of the game can be enjoyed by the user. The board is also connected to a shield (for ECE 385), which contains the USB port needed by the keyboard. A diagram of this is shown below (Fig. 1).

This project is primarily implemented on hardware, and the game logic is at the Register Transfer Lever (RTL) level. We also require the use of a Nios II/e based System on Chip (SoC) in order to use a standard USB keyboard to control the game.

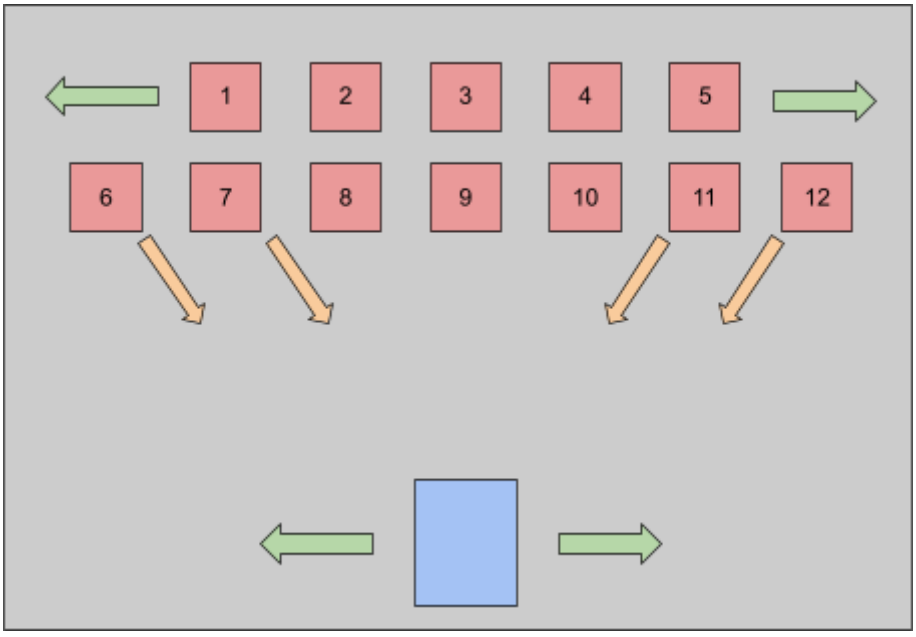
The overall mission of our project was to create a game that was similar to the original Galaxian arcade game. For a high-level overview, the game consists of 13 characters which are displayed as sprites. Of these 13 characters, 12 of them are the enemy aliens and one of them is the player's spaceship. This game, like the original Galaxian, is a fixed shooter arcade game and the player's goal is to shoot down all 12 aliens, while at the same time not being shot themselves. The player's ship can move left and right, and these motions are controlled by the keyboard. The aliens can move side-to-side or diagonally. Both the player and aliens can shoot at each other. If the player is killed by an enemy alien, it is game over. Conversely, if the player shoots all the enemy ships, then the game ends, but this time in victory. Once the game is over, it can be played over again till when the user is satisfied and wants to stop.

Figure 1: DE-10 Board Setup



Gameplay

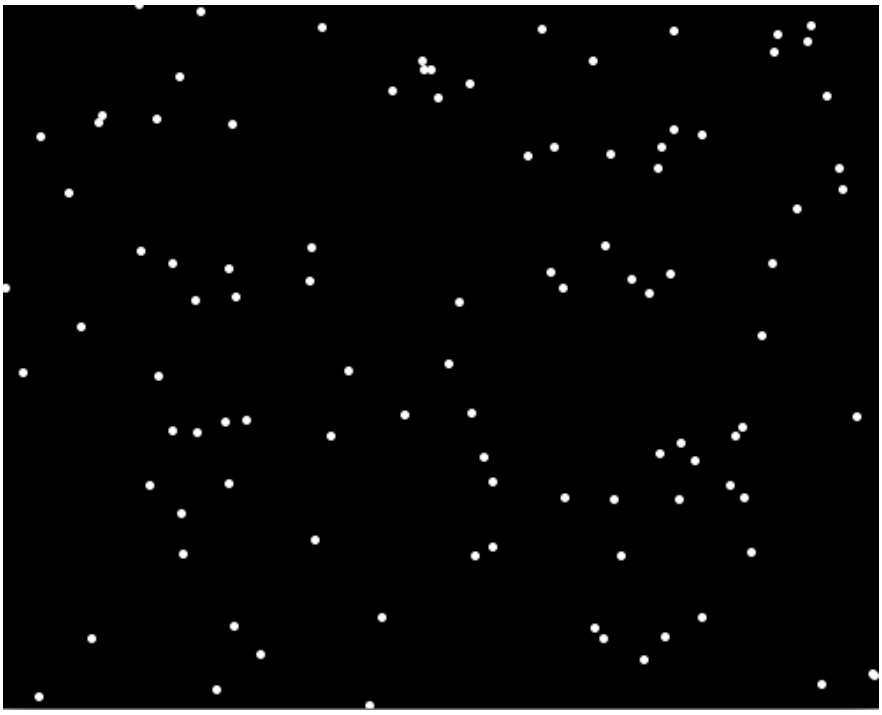
Figure 2: Game Setup



The setup of our game is described in Figure 2. The twelve red squares represent the aliens. The Blue rectangle on the bottom depicts the player ship. The player ship moves from side to side, and most of the alien ships also move in this same way. Initially all the alien ships will move from side to side. However, after two have been shot down, four aliens will move diagonally in succession of each other. When enemy aliens move diagonally, they will follow the player ship's movement. These aliens (6,7,11,12) will also fire missiles at the player ship. The player ship also moves from side to side. This motion is controlled by a USB keyboard. The A and D buttons, along with the right and left arrows, are used to control how the player moves. The player's missiles can be fired with the spacebar. The player's missile goes from the bottom to the top of the screen. When the missile goes all the way to the top of the screen, it returns to its starting position and is made invisible. The missile only becomes visible once the player presses the spacebar. If the player's missile hits an enemy alien, it behaves in the same way as if it were at top of the screen, and it returns to the player's coordinates. The missile fired by the alien is very similar to the player missile in its behavior. The only difference is that it goes downwards rather than upwards.

In addition to what is shown in Figure 1, the game also has a starfield. The starfield from the original galaga game is shown below.

Figure 3: Galaga Starfield

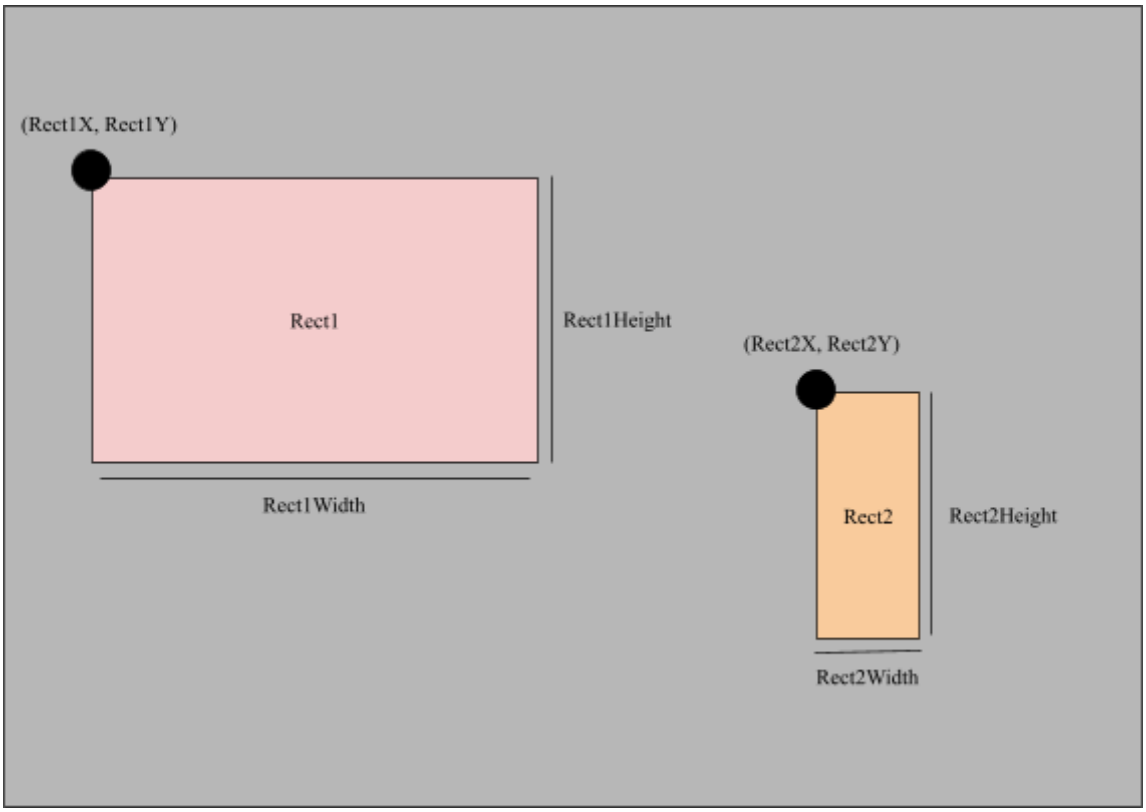


Unlike the original game, the starfield in our game does not contain stars whose positions are random (more on this later). Rather, the starfield consists of 120 stars that are arranged in rows and columns, with slight offsets in each row. In our version of the game, we tried to use the starfield creatively. For example, the colors of the stars would change when the aliens started attacking the player. If the player lost or won the game, the star field would also change. If the player won, the star-field would draw the letter W in yellow, with a purple background. When the player lost, the letter L would be displayed in red, with a white background.

Collision Detection:

Throughout this project we had to determine whether two objects collided with each other. We would have to check whether aliens collide with player missiles, alien missiles collide with the player, or the player and alien missiles collide with each other. We used squares or rectangles for the game entities (player, aliens, etc.) and sprites were imposed onto these rectangles. In order to detect collision, we used a standard approach for determining whether two rectangles collide with each other. Figure 4 describes such an approach. Note that the points are at the top left corner of the rectangles.

Figure 4: Collision of Two Rectangle



The following pseudocode will be used to perform Rectangle-Rectangle collision that we use in this game. This if statement will work as a collision for all languages.

if ((Rect1X + Rect1Width > Rect2X) && (Rect1X < Rect2X +Rect2Width)
&& (Rect1Y + Rect1Height >Rect2Y) && (Rect1Y < Rect2Y + Rect2Height))

Galaxian on Hardware

This game was implemented entirely in hardware at the RTL level. As such, the bulk of our work consisted of creating SystemVerilog modules. We primarily used the behavioral style of HDL coding, and always_ff and always_comb blocks were used extensively. The following modules were used by us to capture more of the game's functionality: *missile.sv, player.sv, alien_missile.sv, simple_alien.sv, control_unit.sv.*

Gameplay Modules:
Module/File: player.sv
Inputs:

input Reset, frame_clk,
input [7:0] keycode,
input am1_sight,
input am2_sight,
input am3_sight,
input am4_sight,
input am5_sight,
input am6_sight,
input am7_sight,
input am8_sight,
input am9_sight,
input am10_sight,
input am11_sight,
input am12_sight,
input logic [9:0] Alien1MissileX, Alien1MissileY, Alien1MissileS,
input logic [9:0] Alien2MissileX,Alien2MissileY,Alien2MissileS,
input logic [9:0] Alien3MissileX,Alien3MissileY,Alien3MissileS,
input logic [9:0] Alien4MissileX,Alien4MissileY,Alien4MissileS,
input logic [9:0] Alien5MissileX,Alien5MissileY,Alien5MissileS,
input logic [9:0] Alien6MissileX,Alien6MissileY,Alien6MissileS,
input logic [9:0] Alien7MissileX,Alien7MissileY,Alien7MissileS,
input logic [9:0] Alien8MissileX,Alien8MissileY,Alien8MissileS,
input logic [9:0] Alien9MissileX,Alien9MissileY,Alien9MissileS,
input logic [9:0] Alien10MissileX,Alien10MissileY,Alien10MissileS,
input logic [9:0] Alien11MissileX,Alien11MissileY,Alien11MissileS,
input logic [9:0] Alien12MissileX,Alien12MissileY,Alien12MissileS,

Outputs:

output logic player_hit,
output [9:0] PlayerX, PlayerY, PlayerS

Description and Purpose:

This module is used to define the player ship in our game. It must keep track of keycodes because when the user presses either A/D of the arrows, the player is supposed to move. The player ship must also keep track of the 12 alien missiles, in case it is hit by any one of them. The output visible tells the color mapper that the alien is visible and that it must be drawn on the screen. The coordinates and size of the player must also be output because other modules need this information in order to be fully functional. Finally, the player module must also know when the alien missiles are visible. This is to avoid a very subtle bug, which is that when the player ship collides with an invisible missile it would disappear. This is clearly erroneous, so we need to give information to the player ship about the visibility of alien missiles. This module is perhaps the most significant module of the entire game since it defines the behavior of the user.

Module/File: simple_alien.sv

Inputs:

input Reset, frame_clk,
input [9:0] alienStX, alienStY,
input [9:0] PlayerMissileX, PlayerMissileY, PlayerMissileS, PlayerShipX,
input [1:0] motion_code

Outputs:

output visible,
output [9:0] AlienX, AlienY, AlienS

Description and Purpose:

This is the module that describes the behavior of the 12 aliens. It was named "simple_alien" because we wanted to create another class of aliens, but did not have the time to do so (more on this later). An alien requires a starting set of (x,y) coordinates, and this information is provided to it as inputs. The module also requires information about the Player Missile. If the alien collides with the player's missile, then it must be made invisible, and this is why there is an output called visible. There is also an input called motion code. There are three possible motion codes that aliens can have: 0, 1 and 2 (in binary). When the motion code is 0, the alien will move to the right. When the motion code is 1, the alien will move to the left. When the motion code is 2, the alien will move diagonally in a way that follows the player ship. In order to follow the player ship, the alien needs to know its x-coordinate. This is why PlayerShipX is an input to this module. Finally, the module needs to output its X and Y positions so that other modules can use this information. This module and the player module encapsulate the functionality of the main two elements of this game: the player and the enemies.

Module/File: missile.sv

Inputs:

input Reset, frame_clk,
input [7:0] keycode,
input [9:0] BallX,
input [9:0] Alien1X, Alien1Y, Alien1S,
input [9:0] Alien2X,Alien2Y,Alien2S,
input [9:0] Alien3X,Alien3Y,Alien3S,
input [9:0] Alien4X,Alien4Y,Alien4S,
input [9:0] Alien5X,Alien5Y,Alien5S,
input [9:0] Alien6X,Alien6Y,Alien6S,
input [9:0] Alien7X,Alien7Y,Alien7S,
input [9:0] Alien8X,Alien8Y,Alien8S,
input [9:0] Alien9X,Alien9Y,Alien9S,
input [9:0] Alien10X,Alien10Y,Alien10S,

input [9:0] Alien11X,Alien11Y,Alien11S,
input [9:0] Alien12X,Alien12Y,Alien12S,
input [9:0] AlienMissile1X, AlienMissile1Y, AlienMissile1S,
input [9:0] AlienMissile2X,AlienMissile2Y,AlienMissile2S,
input [9:0] AlienMissile3X,AlienMissile3Y,AlienMissile3S,
input [9:0] AlienMissile4X,AlienMissile4Y,AlienMissile4S,
input [9:0] AlienMissile5X,AlienMissile5Y,AlienMissile5S,
input [9:0] AlienMissile6X,AlienMissile6Y,AlienMissile6S,
input [9:0] AlienMissile7X,AlienMissile7Y,AlienMissile7S,
input [9:0] AlienMissile8X,AlienMissile8Y,AlienMissile8S,
input [9:0] AlienMissile9X,AlienMissile9Y,AlienMissile9S,
input [9:0] AlienMissile10X,AlienMissile10Y,AlienMissile10S,
input [9:0] AlienMissile11X,AlienMissile11Y,AlienMissile11S,
input [9:0] AlienMissile12X,AlienMissile12Y,AlienMissile12S,
input player_hit,

Outputs:

output visible,
output [9:0] MissileX, MissileY, MissileS

Description and Purpose:

This is the module for the player's missile. The high-level functioning of the missile is that it is invisible and moves along with the player until and unless the space bar is pressed. Upon a press of the spacebar, the missile will become visible and move vertically upwards. Its motion, at this time, is no longer coupled to the movement of the player. Because of this functioning, the missile module must have information on the keycode. It must also know the positions of all of the aliens on the screen, because if there is a collision the missile must go back to the spaceship and turn invisible. The missile needs to also keep track of all alien missiles as well. This is a subtlety, but there could be a scenario where two missiles collide with each other. In this event, both missiles should become invisible and return to their respective starting positions. The outputs of the module are its visibility, which is given to the color mapper, and its coordinates (and size) which is information needed by other modules, such as simple_alien.

Module/File: alien_missile.sv

Inputs:

input Reset,frame_clk,
input shoot_missile,
input alien_hit,
input [9:0] AlienX, AlienY,
input [1:0] motion_code,
input [9:0] PlayerX,PlayerY,PlayerS,
input [9:0] PlayerMissileX,PlayerMissileY,PlayerMissileS,

Outputs:

output [9:0] AlienMissileX, AlienMissileY, AlienMissileS,

output visible

Description and Purpose:

This module describes the alien missiles in the game. There are 12 alien missiles in the game -- one for each alien -- and they behave nearly identically to the player's missile. The key difference is that the keycode is not an input, whereas motion_code is an input. This is because the user cannot control the enemy missiles. The alien missiles must keep track of the player ship, and the player missile, so that the proper behavior can be applied to it on collision.

Control Unit and Finite State Machine

Figure 5: Control Unit RTL Block

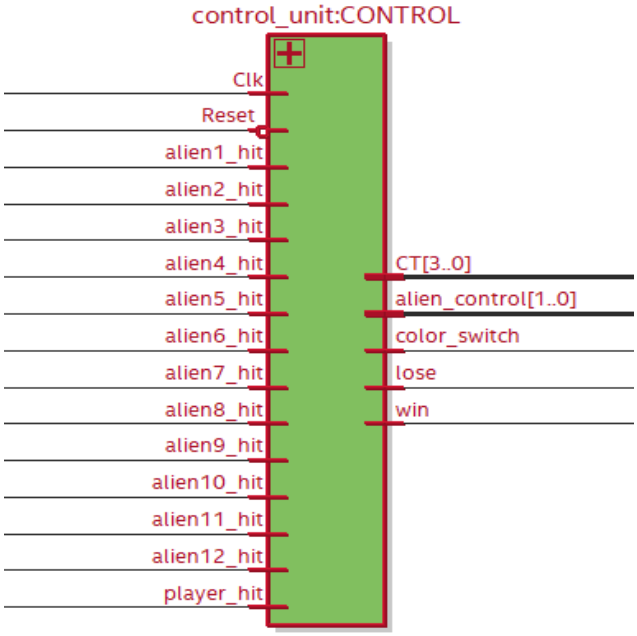
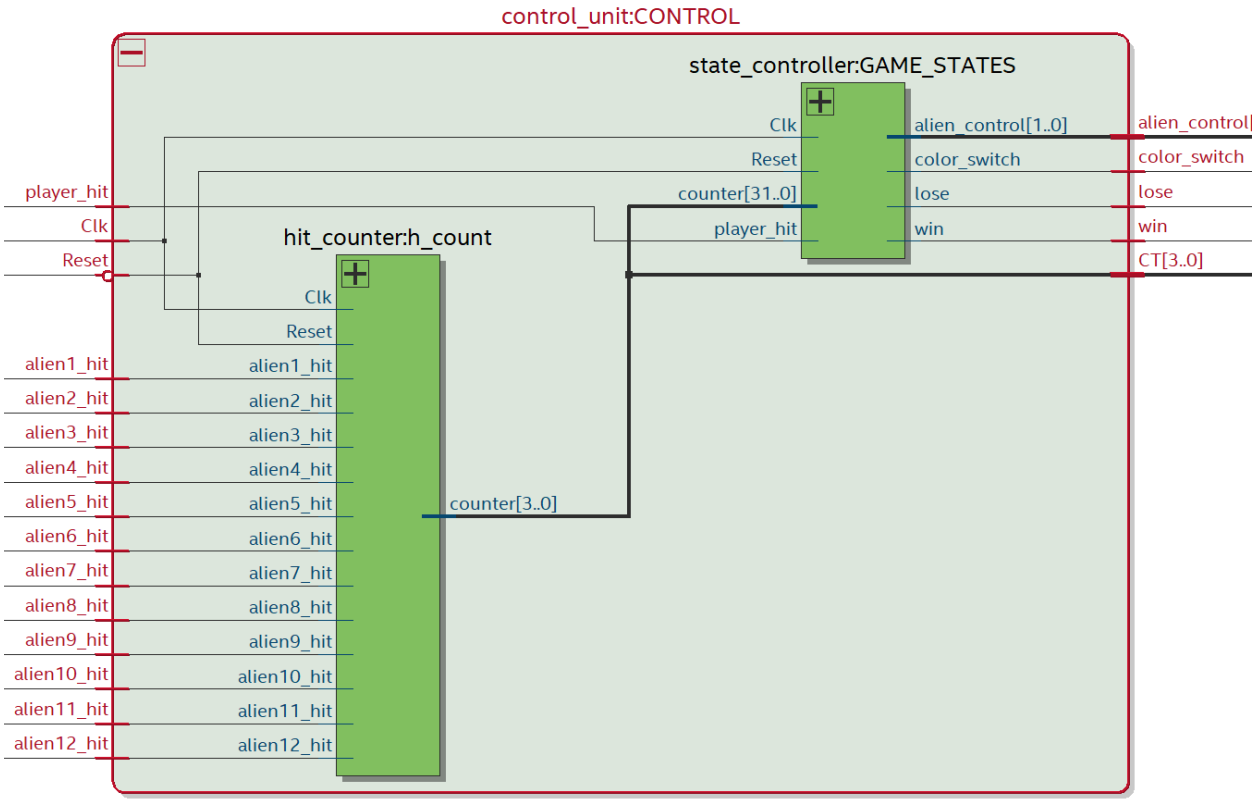
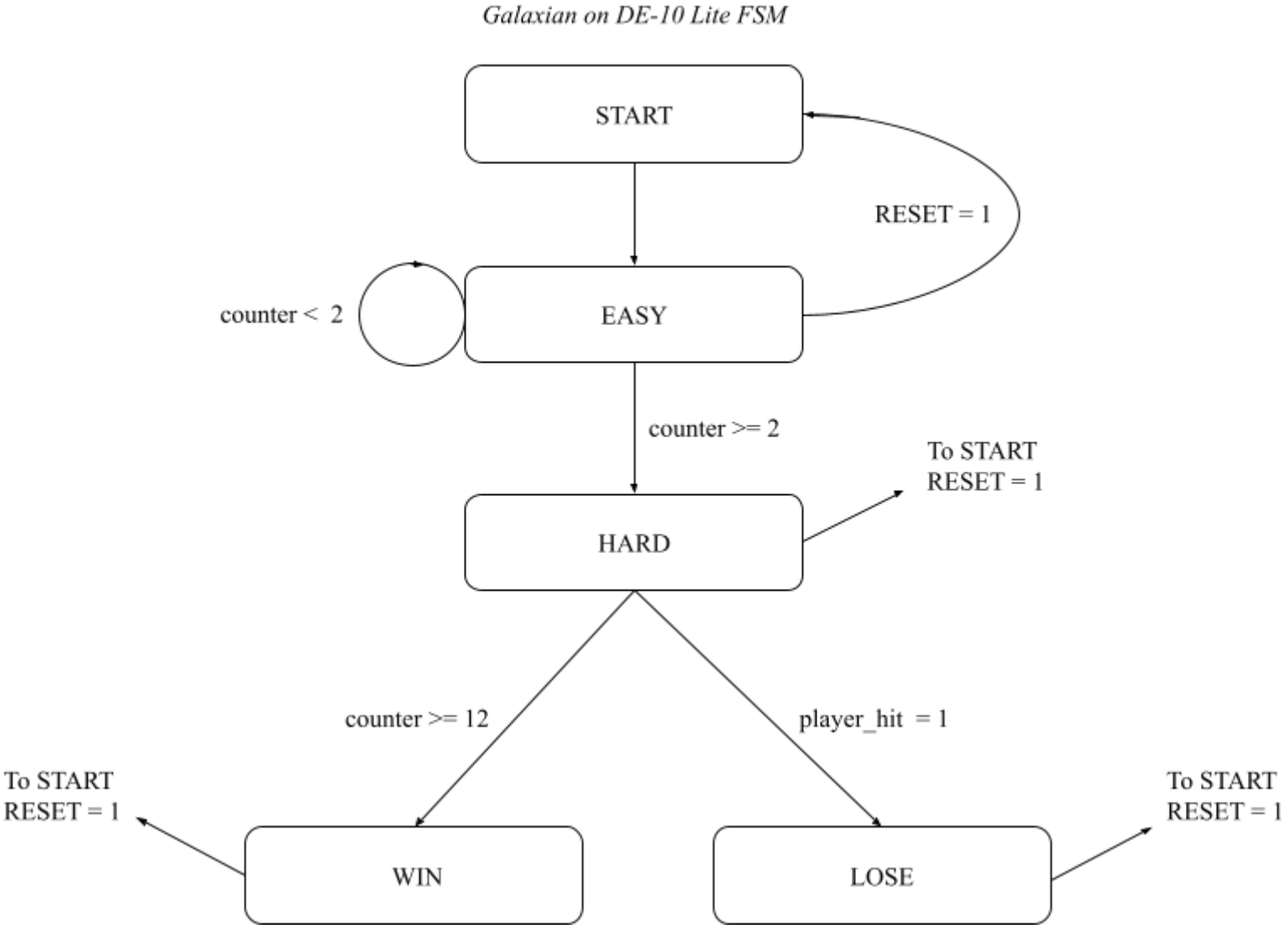


Figure 6: RTL Modules Within Control Unit



Figures 5 and 6 show the RTL view of the control unit. The control unit is what determines the game states and the transitions between those states. This is controlled by a finite state machine (FSM) where there are five states: START, EASY, HARD, WIN and LOSE. The FSM lives in the state_controller module. The FSM for "Galaxian on the DE-10 Lite" is shown in the figure below.

Figure 7: Galaxian FSM



As is shown in Figure 7, the game states are determined based on three signals: Reset, counter, and player_hit. When Reset is high, the game goes back to the start state. The reset signal can be pressed at any point in the game, and the game can be played over and over again. The signal player_hit tells whether or not the player has been shot by an alien. If the player gets shot, its game over, and the game transitions to the game over state. This leaves the signal counter, which controls the state transitions from the EASY to HARD and HARD to WIN. As

shown on Figure 6, counter is the output from a module called hit_counter. The hit_counter module counts the number of aliens that are visible on screen. The output of hit_counter starts off at zero and is incremented when either an alien is shot by the player, or has moved off the screen. Both modules in the control unit, as well as the overall control unit are described below:

Module/File: control_unit.sv
Inputs:

```
input logic Reset,
input logic Clk,
input logic player_hit,
input logic alien1_hit,
input logic alien2_hit,
input logic alien3_hit,
input logic alien4_hit,
input logic alien5_hit,
input logic alien6_hit,
input logic alien7_hit,
input logic alien8_hit,
input logic alien9_hit,
input logic alien10_hit,
input logic alien11_hit,
input logic alien12_hit,
```

Outputs:

```
output logic color_switch,
output logic win,
output logic lose,
output logic [1:0] alien_control,
output logic [3:0] CT
```

Description and Purpose:

The RTL view of this module can be seen in Figure three, and this module is an encapsulation of the two modules seen in Figure 6, and described below. The reason that we encapsulated hit_counter and state_controller into this module was so that it would be easier to test both of the modules at the same time with the same testbench (more on this later). This module is responsible for having the game actually function as a game, rather than a visual demonstration of the game elements. To the effect, the module contains both the hit counter and state controller modules, which act in concert in order to ensure that the transition between game states occurs at the correct time (ie the alien count).

Module/File: hit_counter.sv
Inputs:

input logic Reset,
input logic Clk,
input logic alien1_hit,
input logic alien2_hit,
input logic alien3_hit,
input logic alien4_hit,
input logic alien5_hit,
input logic alien6_hit,
input logic alien7_hit,
input logic alien8_hit,
input logic alien9_hit,
input logic alien10_hit,
input logic alien11_hit,
input logic alien12_hit,

Outputs:

output logic [3:0] counter

Description and Purpose:

This module is used to keep track of the number of times there is a "hit". In this game, a hit is defined as high when an alien has been shot by the player, by its missile, and when the alien moves off the game's screen. When aliens are not hit, then hit is defined to be low. The hit counter needs to keep track of whether all aliens are hit or not, which is why all 12 alien_hit signals are inputs to the module. The three bit signal for the value of the hit-counter is an output because this signal has to go to the state controller in order to dictate when the state transitions occur.

Module/File: state_controller.sv

Inputs:

input logic Reset, Clk,
input int counter,
input logic player_hit,

Outputs:

output logic win,
output logic lose,
output logic [1:0] alien_control

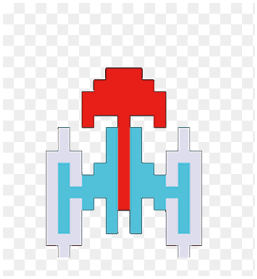
Description and Purpose:

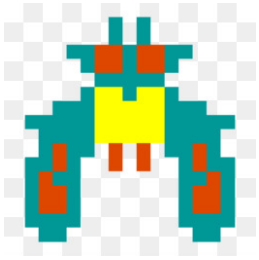
This module is one of the most important in the entire project. This project is an arcade game, and because it is a game it requires a logic flow. That is, there must be clearly defined modes of play as well the ability to know three important details: if the player has won the game, if the player has lost the game, and how to restart the game in a seamless fashion. This module achieves all three of these objectives because it is a finite state machine. More details for this are

given in Figure 7 and the explanation succeeding it. There are three outputs to the FSM which are of importance to the rest of the function of the game. First is the alien control signal, which is important because the signal transitions from 0 to 1 when the state is HARD. The alien control signal is important in the game logic because it is a way to distinguish between behaviors of states. In the EASY state, a motion code of 2 is not used, but when the state transitions from EASY to HARD, a motion code of 2 is permissible. The other two outputs are of equal importance. When *win* is high, we want the starfield to be purple, and to display a W with yellow stars. In this same way, when the player is hit by an enemy missile we want the entire game to be cleared of all aliens and stray misiles. This happens when *lose* is high, and the letter L is drawn on the starfield.

Drawing Sprites on the Screen

Throughout this project, a big part of making our game visually appealing was using sprites for the player's ship and the aliens. The course staff provided us with Python scripts that create sprites from PNGs via PNG to Hex conversion. We used On-Chip Memory rather than SRAM for all graphics. Here are the images we used for the player ship and aliens in our game.





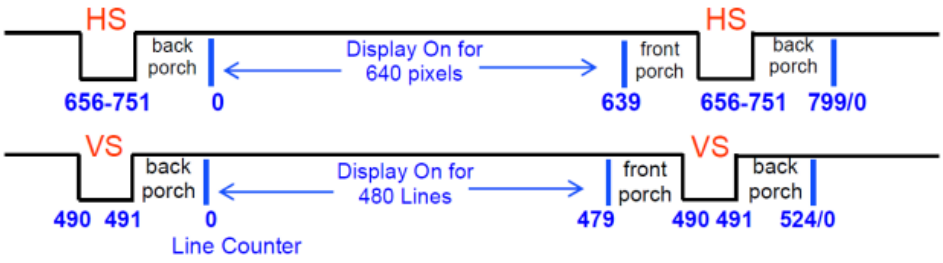
In order to convert the PNGs into sprites we have to first run the provided *png_to_txt.py* script. We have to then save the newly created text file into a folder called "sprite_bytes" in the same directory that our project is in. The provided python scripts also come with a SystemVerilog file called ram.sv. Ram.sv had to be instantiated for each PNG we wanted to turn into a sprite. We also needed a separate ram.sv file for each image. This is why we use two modules, *ram.sv* and *alien_ram.sv*.

Color Mapper and VGA Interface

We used the VGA port on the DE-10 Lite board to connect it to a VGA monitor. Two files were provided to us, *VGA_Controller.sv* and *Color_Mapper.sv*. Of these two files, the VGA controller was left unchanged, but the color mapper was modified by us extensively. The VGA controller is responsible for ensuring that the timing signals for the VGA monitor's operation are created. The VGA display requires two signals, the horizontal signal (HS) and vertical signal

(VS). These two signals are active low for most VGA monitors. Depictions of these signals, shown to us in lecture are given below.

Figure 8: VGA VS and HS signals

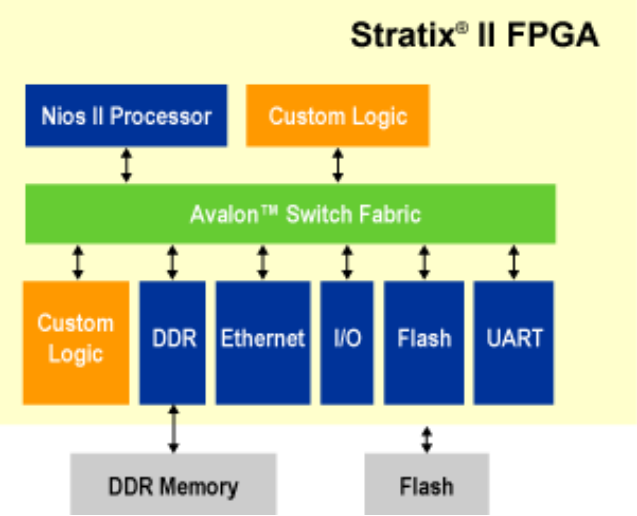


VGA monitors draw pixels onto the screen by way of an "electron beam". Starting at the top left corner of the screen, the beam will draw pixels on each row, from left to right. The screen dimensions of a VGA monitor are 640 x 480 pixels. The color mapper is responsible for sending RGB signals as outputs to the monitor. The color mapper is where all of the shapes are assigned their colors, and where sprites are displayed, rather than shapes.

Nios II/e Based System on Chip (SoC)

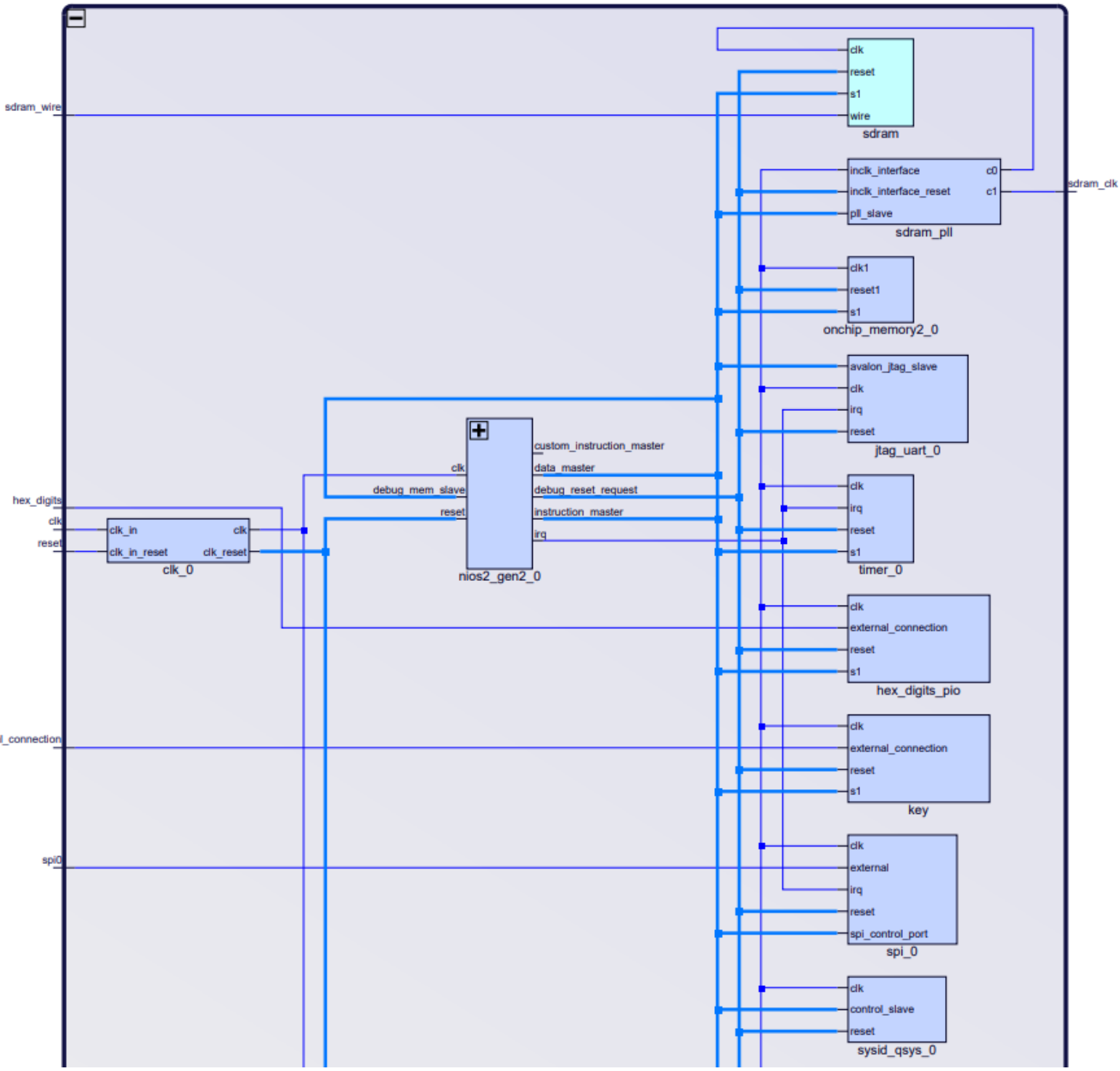
For this project, we used the 32-bit Nios II/e processor to create a SoC to control an external USB keyboard. This enabled us to control the player ship, allowing us to play the game and letting the FSM transition states. Generally, in order to carry out a project in the most efficient manner, there must be both hardware and embedded software development. The hardware is primarily designed in a HDL, such as Verilog, SystemVerilog, or VHDL, and used to implement custom hardware for tasks that are high performance. Software is primarily written in C or C++, and is used for low performance tasks. In our project, the game logic and all game modules, control modules, and display related modules are implemented as hardware. Only the interactions with the USB controller and the Nios II/e are done in Software. The Nios II is Intel's line of embedded softcore CPUs, and the Nios II/e is the free version of this processor that we have access to, based on our version of Quartus Prime. The Nios II is not a physical processor, like an ARM cortex. Rather, it is proprietary HDL code that is instantiated as a module and run on the FPGA. This means that the Nios II is run entirely on the MAX 10 FPGA, and this is why it is referred to as a softcore processor. Below is an image of a sample Nios II Based System.

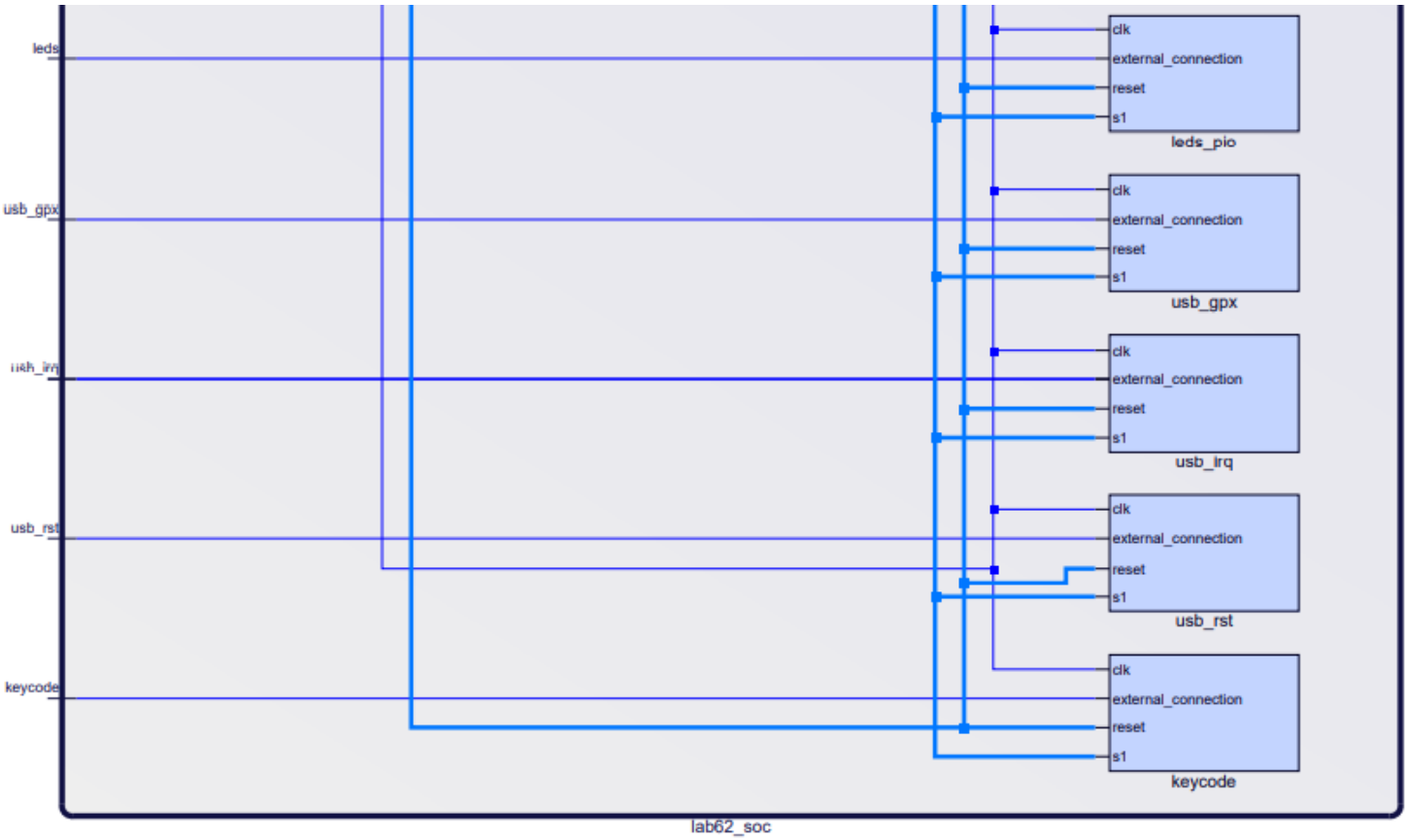
Figure 9: Sample Nios II Based System
FPGA BLOG: The Secret of NIOS II Success (fpgaforum.blogspot.com)



From this picture, there are two key pieces of information to take note of. First, the Nios II is part of a system that includes other pieces of custom hardware. Second, all parts of the picture interact with the Avalon Bus. The Avalon Bus is an interface that allows the user to connect various different components within the Intel FPGA. We will use the avalon bus in our keyboard interface. Below is our system level block diagram from Quartus Platform Designer.

System Level Block Diagram





Functionality of Each Block

clk_0 - Clock Source
This block provides the common clock for the NIOS II/e, and its peripherals, such as the MAX3421E.

nios2_gen_0 - NIOS II Processor
This block represents the NIOS II/e CPU that will be flashed to the MAX 10 FPGA. The Nios II/e handles all software execution. The CPU is connected to the other components of Galaxian since there are common connections, such as a clock.

onchip_memory2_0 - RAM
This block is the on-chip memory. For this project, software C code was executed from here, and these sprites were stored in on chip memory. Specifically the sprites for the player ship and aliens.

sdram - SDRAM Controller Intel FPGA IP

For the project, we used sdram to store the software programs. This component's function is to interface the sdram with the Avalon Bus.

sdram_pll - ALTPLL Intel FPGA IP
This PLL component provides a clock signal for the sdram. Since the sdram requires precise timings, sdram_pll is needed to regulate the clock. This block has two clocks. One clock is set to 50 MHz and goes out to the sdram controller, while the other goes out to the SDRAM chip itself.

sysid_qsys_0 - System ID Peripheral Intel FPGA IP
This block is a system ID checker whose purpose is to verify the compatibility between hardware and software. The component gives a serial number that the software checks against. This ensures that the software being loaded is not being loaded onto an FPGA with an incompatible NIOS II configuration.

jtag_uart_0 - Serial
This block allows the terminal of the host computer to communicate with NIOS II/e. For our project, the terminal is the eclipse console. This block allows us to use printf statements from NIOS II through the USB. It is primarily helpful for debugging software code.

timer_0 - Interval Timer Intel FPGA IP
This block must be instantiated and connected to clock, reset, data, and other peripherals since it is required by the USB driver code for timing, specifically the time-outs that the USB needs (citation: IUQ.3).

spi_0 - SPI Intel FPGA IP
This block allows for SPI communication. SPI as was said before is how the Nios II/e communicates with the MAX3421E USB host/peripheral controller.

usb_gpx
This is a PIO block that is one of the necessary three needed for a connection to the MAX3421E USB host/peripheral controller.

usb_irq
This is a PIO block that is one of the necessary three needed for a connection to the MAX3421E USB host/peripheral controller.

usb_rst
This is a PIO block that is one of the necessary three needed for a connection to the MAX3421E USB host/peripheral controller.

keycode
The block is a PIO module that is necessary to output the keycode of a button whenever it is pressed. It is particularly useful to us because we require the use of the A/D and the arrow keys for this project to work.

key
This block is a PIO module that represents the 2 push-button keys KEY 0, and KEY 1. Its data width is 2 bits and its direction is input.

leds_pio
This block is a PIO module that is necessary for the values to the leds to be outputted.

hex_digits_pio
This block is a PIO block module that is needed for the value of hit counter to be loaded onto the Hex Displays of the board

Soc System Verilog Module

Once we made a design in Platform Designer, we had to generate the HDL that would contain all the features of the SoC we just designed. The generated Verilog module description for the SoC is given below.

Module/File: lab62_soc.v
Inputs:

```
input  wire clk_clk,
input  wire [1:0]  key_external_connection_export,
input  wire reset_reset_n,
input  wire spi0_MISO,
input  wire usb_gpx_export,
input  wire usb_irq_export,
```

Outputs:

```
output wire [15:0] hex_digits_export,
output wire [7:0]  keycode_export,
output wire [13:0] leds_export,
output wire sdram_clk_clk
output wire [12:0] sdram_wire_addr
output wire [1:0]  sdram_wire_ba
output wire sdram_wire_cas_n
output wire sdram_wire_cke
```

```
output wire sdram_wire_cs_n
inout  wire [15:0] sdram_wire_dq
output wire [1:0]  sdram_wire_dqm,
output wire sdram_wire_ras_n,
output wire sdram_wire_we_n,
output wire spi0_MOSI,
output wire spi0_SCLK,
output wire spi0_SS_n,
output wire usb_rst_export
```

Description and Purpose:

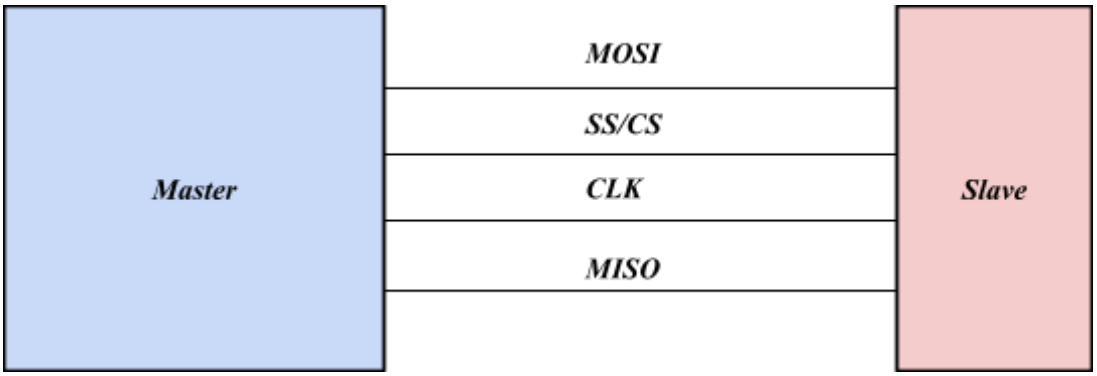
This module is responsible for connecting all of the various SoC components in order to enable SPI communications with the keyboard. These are the PIO blocks shown above, along with Nios II/e. The module, generated by the platform designer, contains all of the necessary inputs, outputs , and PIO Blocks to communicate with the MAX3421E. In particular these are the usb_rst, usb_gpx, usb_irq, keycode, and spi_0 blocks.

Keyboard Interface

As stated earlier, our DE-10 Lite board is connected to the ECE 385 Shield, a custom PCB designed by the course staff, via its Arduino-based pins. This shield has a USB port on it that is connected to a keyboard. On the shield sits the MAX3421E chip, which is a USB peripheral/host controller made by Maxim Integrated. The MAX3421E communicates with the Nios II/e through the SPI protocol. The Nios II/e is the host device, while the MAX3421E is the peripheral device. For the final project, we used the *four functions* that we wrote for Lab 6.2, and these functions allowed for SPI transitions with the device registers on the MAX3421E. SPI communication to and from the MAX3421E is possible because the Avalon Bus was utilized.

The SPI Protocol is a serial, synchronous communications protocol between two or more devices. There are four SPI Connections: MOSI, MISO, CS/SS, and CLK. These stand for Master Out Slave In, Master In Slave Out, Chip Select/Slave Select and Clock. In our setup the Nios II/e is the master device and the slave device is the MAX3421E. Additionally, SPI is known as a synchronous communication protocol because all connected devices operate off of a common clock. Figure 10 shows a sample SPI connection.

Figure 10: Sample SPI Setup



In order to have the Nios II/e communicate with the MAX3421E, we used to write four functions, and this was done by us in Lab 6.2. These functions were written in C, and they used the Avalon Bus to perform single and multiple byte SPI reads/writes. Two of the functions used the Avalon Bus to perform a single byte SPI read/write, to/from a MAX3421E device register. The other two were very similar and used to perform multiple byte read/writes into a divide register. The four function signatures are given below.

- ***void MAXreg_wr(BYTE, BYTE);***
- ***void MAXreg_rd(BYTE);***
- ***void MAXbytes_wr(BYTE, BYTE, BYTE*);***
- ***void MAXbytes_rd(BYTE, BYTE, BYTE*);***

In order to use the Avalon Bus to perform SPI transactions we had to use the ***alt_avalon_spi_command*** function, which is documented by Intel.

Testing and Debugging

SystemVerilog is a powerful HDL that is useful for hardware design and verification. The power of SystemVerilog over Verilog is in its testbench features. There are many advanced features of SystemVerilog that exist specifically for testbench development. These features are widely used in industry by hardware verification engineers. We used a testbench for this project as well, and although it was quite basic, it aided us in the debugging process. One area where a testbench was particularly helpful, was in the control unit. We wanted to make sure that the hit counter and state controller were working together. We needed to verify whether the state transitions happened based on what the output of the hit counter was. Creating a testbench and viewing the waveform on Model-Sim was very helpful in testing this part of this code. Below are some of our simulation results. Figure 11 shows the state transitions if the player were to win the game. Figure 12 shows the state transitions if the player were to lose the game.

Figure 11: Simulation of a Win

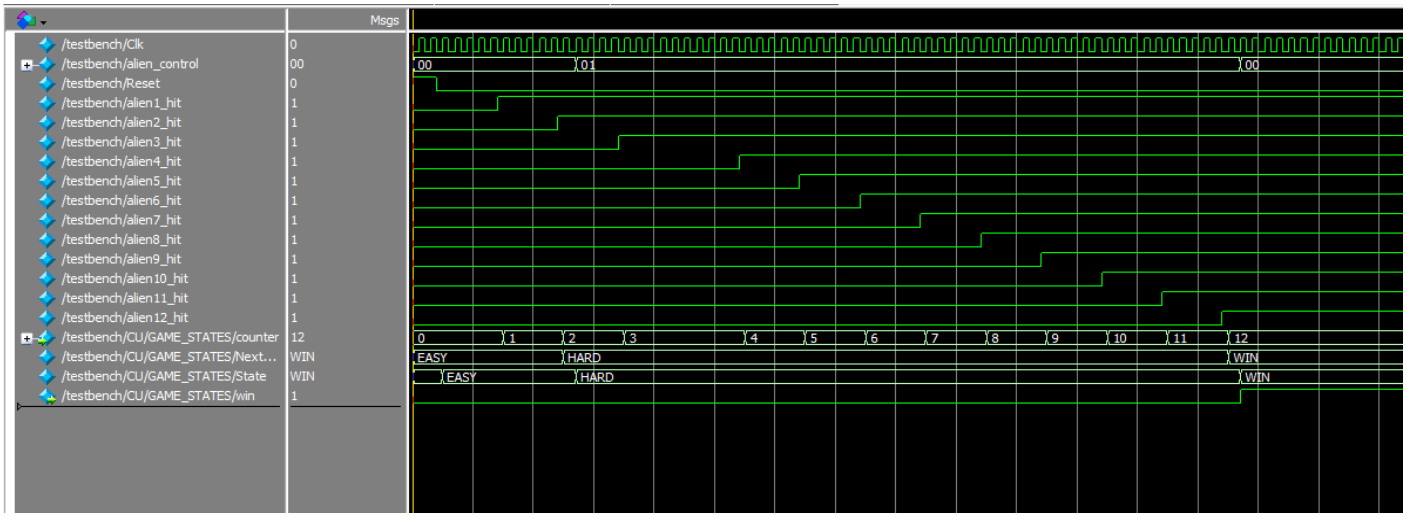
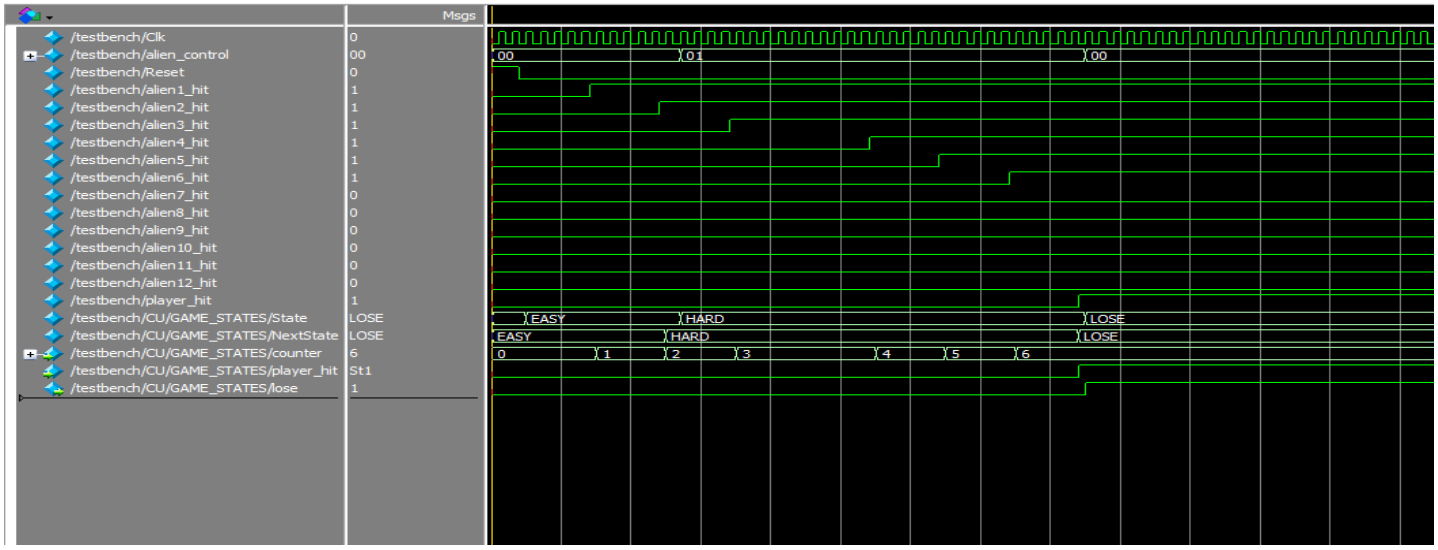


Figure 12: Simulation of a Loss



Improvements and Extensions

If other groups want to work on this project, or recreate it we have a few suggestions for worthwhile project extensions. First, we feel that this game should have sound. When the spacebar is pressed, or in state transitions, sounds would add to the game in a positive way. If we had added sound to our project, the SGTL5000 which was on our DE10 Lite Shield. An I2C peripheral would also have been used, by adding it to the Platform Designer.

Nearly all arcade games were controlled by some form of a joystick, so joystick support would be a much welcomed addition to this project. In order to do this we would have to use an ADC provided to us on the board. In this vein, adding support for a USB game controller, such as a PS4 or Xbox controller would enhance the user experience of the game.

The game would also benefit from more aliens being present. We initially wanted to have two classes of alien: simple_alien and complex_alien. The complex_alien would be bigger in size, and would be harder to shoot than the simple alien; it would also move faster too. We were not able to get around to implementing this feature but we feel it should be in the game. We also feel like some degree of randomness would make the game interesting. In software, the C standard library has methods to take care of this, but if this were to be done on hardware, a linear feedback shift register (LFSR) would have to be used. The original arcade Galaga had an LFSR chip to act as a pseudorandom number generator. Randomness could be used to determine which aliens move diagonally, and which stars are visible in the starfield.

Finally, this game is only operable on Altera/Intel based FPGAs since it utilizes a Nios II Based SoC. In the FPGA market, the big players are Altera and Xilinx, which are owned by Intel and AMD (expected end of 2021) respectively. A user could run this game on a Xilinx FPGA

development board, such as the Arty A7 by Diligent. There would be no differences in the SystemVerilog code, with the exception of the top-level design file. The only real difference would be in the SoC component of the project. Instead of a Nios II/e based SoC, the user would have to use a Microblaze based SoC because that is the soft-core processor available on Xilinx FPGAs.

Conclusions

The successful completion of this final project marks the end of our time in ECE 385. This project was the culmination of a semester's worth of hard work, where we learned industry relevant skills all the while. This project was one of the most unique experiences for us in the U of I ECE program because it was the first hardware based project we had done. Unlike other final lab assignments in other classes, this project gave us the freedom to make whatever we wanted. In doing so, we were faced with many challenges, such as coming up with an initial idea, creating a project structure, working together remotely, and debugging elements that did not work. Working together remotely and debugging out code were particularly challenging because both partners in this group are in different locations, and with respect to debugging, we had to resort to extensive trial and error. These challenges ended up being minor snags, as we were able to resolve them through tools such as GitHub, and a better understanding of testbenches.

Ultimately, this project was a positive learning experience for us, because it led us to appreciate the intricacies of hardware design, and it also allowed us to think out of the box and be creative, while still in an academic setting. In reflection of this project, we feel that we are better prepared to be engineers now than we were prior to starting it, and we will remember this project fondly in the future.